A GOAL MODEL FOR DYNAMIC SYSTEMS

by

MATTHEW MILLER

B. S., University of Nebraska at Kearney, 2003

THESIS

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2007

Approved by:

[illegible]

Major Professor
Scott A. DeLoach, Ph.D.

ABSTRACT

Goal based systems are well suited for creating systems in which there is a well-defined set of goals. These goals easily allow the designer specify what the system should do when the goals of the system are static and very clear-cut. A problem with current goal-based systems is that the goals and system invariants are elicited during requirements. These goals are used during analysis of invariants, but these goals and invariants are not used during the runtime of the system. The lack of use of the goals during the runtime leads to one of two options for software designers. They either will not do the analysis because it will not be used later on, or they will invest time working on requirements that are of no use in the runtime. The lack of incentive to use analysis is a problem because analysis is key in reducing errors in software. To help solve these two problems we propose a goal model that both is dynamic in nature and that ensures that properties verified through analysis hold during the runtime of the system. The dynamic nature of the goal model will allow systems to be designed that are more flexible in nature. The model also allows the designer of the system to specify the goals and properties of these goals. Properties that are specified during the design phase can be proven to hold while the system is running. This verification will aide designers in creating better systems.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. Scott DeLoach. He has helped me through this long tedious process. I would also like to thank my wife Beth. She has been patiently waiting for me to finish this project. I also would like to thank my daughter Addison, for she has brought more happiness to our lives.

Chapter 1

1.1 Problem

Software development has evolved significantly from its relatively recent inception. Early software systems were designed to solve a single problem within a single domain. A large amount of time was spent to make the algorithms run efficiently (be it space or time). These systems typically had very limited resources and were designed run on a single computer. These type of systems eventually became very good at solving these problems (and were coined as stovepipe systems), but these systems do not tend to interact well with other systems [13]. Systems developed today need to interact with other systems. The interaction with other systems leads to error prone code that is hard to test and debug. As the number of interactions between systems increases the number of potential errors increases dramatically. The solution to this problem has been to either use formal methods for analysis [16] or provide a framework that abstracts some of the low-level details [14].

A formal analysis is the best approach to use to test a system and ensure that properties of that system are met [7]. There are two problems with formal analysis. The first is that formal analysis requires quite a bit of background knowledge to use. The second is that formal analysis of a large system can take exponential time to compute [22].

Formal methods are often used in practice for the elicitation of software requirements. The elicitation of requirements via formal methods allows for the elicitation of requirements and their gradual refinement into more detailed requirements [14]. In addition to eliciting requirements, this process also helps to derive system invariants that should hold while the system is running. The correctness of these properties can be verified using model-checking software [25,17,3], theorem provers and through restrictive frameworks [7].

The use of formal methods in the elicitation of requirements provides a good mechanism for gathering and understanding the requirements of the system. The use of stepwise refinement can take formally specified high-level requirements and produce correct runtime code [28, 27]. The code produced by these models retains the high-level properties and invariants. However, currently there exists no multiagent specification model that continues the use of the formal models from the elicitation of high level specifications through the runtime of the system. A multi-agent model lies at a higher level than the specification used for stepwise refinement, but similar specifications are usual for specifying the low level modules within the agents themselves. The failure to use formal models throughout the entire process invalidates the verification of properties the system at a high-level. Thus the properties of the requirements model can not be automatically be transferred to the runtime system. However, if a formal model were used throughout the software life cycle, then the resulting software would have the properties elicited in the requirements.

Goal models provide a framework that abstracts away low-level requirements. Goals are a natural way to design multiagent systems, as they provide a key abstraction. They allow agents to work on an abstract goal, while allowing the manner in which goal is achieved to be flexible. This flexibility makes goals a key abstraction in many multiagent systems [20,23,19]. In addition to abstraction, goal models have a well defined semantic meaning. The semantic meaning of a goal model allows the system interactions to easily be understood by the designers [26]. A better understanding of the system interactions clearly should allow designers to design software with fewer errors. Goal models are used during the requirements phase [20], but all of the current goal models are not used directly during system runtime [20,32,1].

A framework that allows end-to-end use of a single goal model would be very desirable if that model could provide additional assurance of correctness. The goals that are elicited during the requirements phase need to be directly translated into goals for use during the implementation and runtime phases. Therefore, I propose to develop a goal model that has precisely defined semantics, manages the complexity of the system, provides a transformation from specification into implementation and runtime models, and allows invariants to be verified at various stages throughout the software life cycle.

1.2 Background

Multi-agent systems are a relatively new field in computer science. When computers were scarce computing time was shared among many users. This situation began to change as computing power has become cheaper, faster and smaller. As computers have become more powerful applications have become more complex and thus system designers must think at higher, more abstract, level and utilize the resources efficiently.

In modern systems, there may be many stakeholders that need to interact in a complex manner in order to share resources. The stakeholders may not be in total control of all of their resources, and thus must rely on other stakeholders to use those resources. The protocols that each stakeholder must partake in are complex, as there may be multiple interactions with multiple stakeholders to use the remote resources. This is similar to multiagent systems where resources are distributed across networks and the interactions between the agents in the systems are complex.

The key abstraction in multi-agent systems is that of the agent. An *agent* is an autonomous entity that works on a specified task. The agent is in charge of deciding how it will play its role in the system [30]. This frees the system designer from knowing all of the details of how a role is played. The designer can assume that the agent knows how to play a role in the system, which allows the designer to think at a higher level. The agent designers need only design the agent to play the role according to the specification given [30]. The system uses its knowledge of the abilities of the agents in order to operate in an efficient manner. The system can assign agents to roles that they are good at playing.

Another key element in the design of multi-agent systems is that of the goal. A *goal* captures a desired state of the environment [15], as perceived by agents in the system. Each goal can be decomposed into multiple smaller goals. This decomposition allows the entire system to be designed by generating the smaller more manageable parts [11]. Goals are used in many current methodologies during the elicitation of requirements [26]. Current methodologies can use formal methods to verify the axioms that have been elicited [33]. These formal methods are applied to the models that are derived from the requirements elicitation process. There is however, no connection between the requirements elicitation and the design and implementation phases in multiagent methodologies.

1.3 Scope and Objectives

The objective of this thesis is to define and formalize a Goal Model for Dynamic Systems (GMoDS). GMoDS must be able to model system-level goals and their relationships. The system designer must be able to specify the overall goal of the system and iteratively decompose it into several subgoals that can be achieved by agents in the system. Multiagent systems are event driven by nature and thus the model must be able to specify the effect of events on system level goals. In order for the application to reason about the events that occur in the environment, the designer must define those events. The events must allow the goal model to dynamically adapt to the changes occurring in the environment. Because multiagent systems are distributed, they require a method for defining an ordering of goals. Therefore, the model should also allow the designer to specify a partial of full ordering of goal achievement [5]. Applications need to use a full ordering if there are goals that need to be sequentially accomplished and either a partial or no ordering if the goals can be accomplished independently. The ability to order goals gives the designer flexibility when specifying the goal model. A restrictive design includes a full ordering, while a less restrictive design requires only a portion of the goals to be ordered. The partial ordering allows the system to choose the order of the goal achievement.

The scope of this thesis is limited to the formal definition of the semantics of the GMoDS model, the translation of the GMoDS model into the runtime model, the semantics of the runtime model, and a proof of concept implementation. The thesis does not cover such things as how the system assigns goals, how an optimal set of assignments is found, or the elicitation of requirements.

1.4 Document Overview

A review of literature in multiagent systems occurs in Chapter 2. The literature reviews the current state of the art in multiagent design and goal modeling in multiagent systems. This section helps to outline the shortfalls of current multiagent systems, such as the lack of an end-to-end goal modeling. The formal definition of the GMoDS occurs in Chapter 3. The GMoDS definition includes the background needed to understand GMoDS. It also includes the formal definitions of the concepts, explanations of those concepts, and examples to help guide the user. Chapter 4 defines the Execution Model that has been developed for use in dynamic multiagent systems. This model provides a concretization of how the goals in the model flow from one state to another state. Chapter 5 presents an implementation of the dynamic runtime execution model. This includes the key portions of the source code along with the challenges that were encountered during the implementation. Chapter 6 describes the results that were gathered from a proof of concept implementation. Chapter 7 describes the conclusions drawn from this thesis and presents future work.

Chapter 2 Literature Review

This chapter briefly reviews what has been done related to goal modeling in multiagent systems and multiagent methodologies. The work in this paper is based on previous work done in the fields of multiagent systems and goal modeling. Multiagent systems have roots in distributed computing, artificial intelligence, software engineering, formal methods and embedded systems. These fields are quite broad, and thus relevant content will be compared to show how this work uses concepts from these fields to bring about improvements. The improvements GMoDS provides should allow for the development of systems in a less error prone environment.

2.1 Goal Modeling

The goal-based abstraction has been commonly used to model specifications in agent-oriented systems. The object-oriented abstraction was the basis for many agent-based methodologies such as *i** and GAIA. A problem is that objects tend to map to real world entities, while goals map to states of the world. Theses states of the world are as observed by agents (specific objects) in that world. This implies two different semantic meanings for the goal-based abstraction and the object-based abstraction. The next section is devoted to the work done on the goal-based abstraction for modeling specifications.

2.1.1 KAOS

The Knowledge Acquisition in autOmated Specification (KAOS) framework was developed for use in the modeling of system requirements [20]. The model of the system requirements is a structured formal model, which can include system invariants and properties. The structure and formality of this model allows a model checker to verify the system invariants and properties.

The KAOS model requires the structure of system goals to form a tree. The goals are specified at a high-level, and then each of the high-level goals is decomposed into a set of sub-goals. This decomposition process continues until the goals cannot be decomposed. Each level of decomposition is either disjunctive or conjunctive. A disjunctive high-level goal is satisfied when a single sub-goal is satisfied, whereas a conjunctive high-goal is satisfied when all of its sub-goals are satisfied. KAOS also specifies that satisfaction of goals can contribute to, or degrade the satisfaction of other goals in the system. These effects occur when agents in the system have different motivations, perceive information differently or when goals interact with one another. The KAOS model gives allows some information to be specified that may enable the system to determine which agents in the system are best capable to work specific goals. The lack of formalization of how the system should determine which agents are best at playing which tasks is a limiting feature of KAOS.

The KAOS model design is closely related to requirements elicitation. When a KAOS model is developed, it is developed in two distinct yet coordinated tasks. The first task is the elicitation of the meta-model. The meta-model specifies system requirements at a high level.. An example of this is in Code Listing 1. The goal is MetaSocketSpec and the system wishes to achieve that goal.

```
Achieve[MetaSocketSpec]
```

Code Listing 1[20]

Requirements elicitation is the most difficult part of software design. Errors in the requirements stage are more expensive to fix than any other errors in the software life cycle. The second task is the acquisition strategy. This is a process that allows the system designer to navigate the meta-model searching for items that need refinement. The refinement allows additional components to be elicited as they are needed. Continuing the example in Code Listing 1, the goal can be AND decomposed into the four goals in Code Listing 2.

```
Achieve[Insecure output never produced],
Achieve[All input packets are output],
Achieve[No packet output before corresponding input arrives],
Achieve[AdaptVar].
```

Code Listing 2[20]

This process allows the meta-model to gain additional information in an incremental fashion. In addition to these two tasks, there is also an assistant to help automate the process of eliciting the requirements. The assistant can pull domain specific knowledge from a database and thus provides reuse of domain knowledge. The reuse of domain knowledge allows the requirements elicitation to progress much quicker when this knowledge exists.

KAOS also allows triggers and actions to be elicited, represented, and refined in the model. Triggers cause the system to add a goal, and an action is a manner to achieve a goal. An example of an action is in Code Listing 3. The action is for checking out a book and the action defines the inputs, outputs, a precondition and a post condition. When the precondition becomes true, then the post condition will be assumed to be True at the end of the code execution.

```
Action CheckOut
   Input  BookCopy {Arg: bc}, Library {Arg: lib}, Borrower {Arg: bor}
   Output Library {Res: lib}, Borrowing
   PreCondition  bc ∈  lib.available
   PostCondition  ¬ (bc ∧ lib.available) ∧  bc ∧  lib.checkedOut
                                 ∧  Borrowing (bor, bc)
```
Code Listing 3[20]

There are several benefits to the meta-model representation. First, typically many different systems use the same abstractions to represent the entities in the system. These abstractions can be reused in many different systems, and therefore will speed up the design of systems. Second, the meta-model allows for high-level reasoning, without knowledge about the domain of the system. The formal meta-model allows model-checkers to reason over the model, and thus check properties of the model. The properties of the meta-model give feedback to the designers, which in turn can be used to elicit more requirements of the system. An example of this would be if the designer had stated that some property A should always hold. If the model checker states can find an trace of the system that violated A, then that case can be analyzed, the failure can be understood, and the model or the property can be redesigned.

2.1.2 KAOS elicitation with model checking

KAOS is an excellent model for doing specifications, but without a tool that provides model checking ability, it can add undue overhead the design of a system. The absence of a tool for automated model checking would require designers to have knowledge of formal methods in order to design the system. There are many model checkers that are capable of doing model checking for KAOS, including SPIN [17], Bogor [25] and SVM [3]. These model checkers can verify the specifications in formal systems such as telephone systems, code verification and protocol designs. Applying model checking to specifications written in the KAOS specification language is beneficial [2]. The KAOS models tend to use Linear Temporal Logic (LTL) or a form of LTL to specify safety and liveness properties. An example of an LTL formula would be as follows in the **FormalDef** section of the specification of the goal. In this formula if S_{SPEC} is true and A_{REQ} is eventually true then T_{SPEC} is true.

Goal *Achieve*[Adapt from DES 64 to DES 128]
Concerns SSPEC, TSPEC, AREQ where
SSPEC = DES64SPEC TSPEC = DES128SPEC
AREQ = Request 64 to 128 OnepointREQ
RefinedTo In DES 64 State ; In DES 128 State
InformalDef The program initially satisfies: SSPEC.
When it reaches a safe state all obligations generated by
SSPEC are satisfied.
FormalDef $S_{SPEC} \wedge \Diamond A_{REQ} \rightarrow T_{SPEC}$

Code Listing 4 LTL example **[2]**

This LTL formula comes from a group that has designed a graphical font-end for eliciting, understanding the formal specifications for a system [1]. These properties are verifiable by any of the model checkers listed above. The verification of the properties ensures that a proper implementation will not violate the specified properties.

2.1.3 KAOS conflict resolution

The process of eliciting requirements in any system tends to lead to conflicts. This is no different when specifying a KAOS model [26]. It is therefore important to resolve these issues at a high level which ensures that the goals, when implemented, do not conflict with one-another. Elicited requirements are often either too loose or too strict. Requirements that are too loose, can be tightened later on in the software process. The overly strict requirements are the ones that lead to goal conflicts. Strict requirements are so, because they tend to assume a granularity that is larger then necessary. The large granularity means that the property must hold for a large amount of states of the system. By decreasing the granularity, the verification of the property is pertinent to only the states where it is required. Overly strict requirements can also lead to resource deadlocks or data inconsistencies, and therefore must be resolved. In [21] conflict analysis was applied to goal specifications using the KAOS methodology. They determined that there were a variety of situations that can lead to goal conflicts including multiple viewpoints, assertion conflicts and goal divergence. Multiple viewpoint conflicts arise when two different stakeholders disagree on the state of the system. Assertion conflicts are found when invariants of the system are fed through a model checker or theorem prover and a counter example is produced. Goal divergence arises when goals in the system can become logically inconsistent at some boundary condition [21].

```
View1: InOperation → Running
View2: InOperation → Startup
       Startup → ¬ Running
View3: □InOperation
```

Code Listing 5[21]

In the example above we can see that each View(1-3) has different requirements and a single View would not be satisfactory. Each of the three parties needs to have a separate viewpoint, as they all perceive a different state of the system. In this refinement we can see that the three Views do not have any high-level conflicts (as in View1 $\leftrightarrow \neg$View2). If there were no conflict resolution, the listing above may either be too weak (View 3) or too strong (View 2) for all parties involved.

They also determined that there could be various approaches to resolving the conflicts. The resolutions include assertion transformations, conflict anticipation, goal weakening, and alternate goal refinement. These resolutions can be used at many different levels of the goal refinement process. The two resolutions that are of particular interest are the goal weakening and the alternate goal refinement. Goal weakening takes a goal that has a coarse definition and makes that goal weaker by either weakening the pre or post conditions of that goal. Alternative goal refinement takes a single goal and produces several specialized goals. These specialized goals are able to handle diverse situations better than the single goal that existed before.

2.1.4 KAOS Summary

KAOS provides a good background for the GMoDS framework. KAOS takes the elicited requirements and creates a model that can create a system. KAOS provides a goal model that has a hierarchical framework with and/or decomposition. KAOS model extensions have provided system designers with the ability to verify system properties and to perform conflict management and conflict resolution. KAOS does not however provide an end-to-end goal modeling framework or a dynamic runtime model. The framework and runtime model are needed to create a viable multiagent modeling solution.

2.1.5 *i** Framework

The *i** framework was specifically designed to for organization based systems [32]. The *i** framework specifies allows the interactions between actors (agents) to be formally specified and therefore reasoned about and possibly model checked. The actors are considered to be autonomous, but are the environment they work in is constrained by the environment and other actors in the system. The other actors in the system may control limited resources. This forces the actors to interact with other actors in order to use the limited resources and to accomplish goals in the system. The actors are not considered to be benevolent, and therefore they must act in a strategic manner to accomplish their goals. The dependencies between the actors are also modeled. The dependency model is known as the strategic dependency model. The strategic dependency model allows the designer to specify the interactions of the different actors in the system. The dependencies are modeled as a graph, where the nodes of the graph represent the actors, and the edges represent the different dependencies. The edges are also labeled in order to provide information about the relationship that is represented. The labels on the edges indicate the type of dependency. In Figure 1 there are four actors represented by circles, and the edges that connect the circles are the dependencies (which are labeled appropriately).

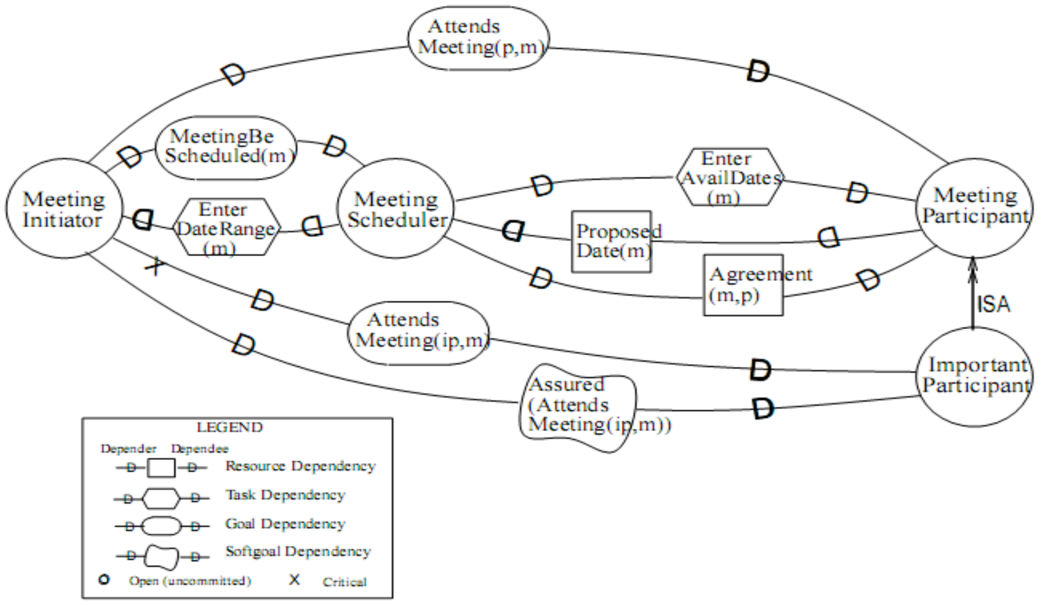


Figure 1. i* Example Model **[32]**

There are four main types of dependencies; resource, task, goal, and soft goal. Resources are items that are of a limited availability. A task is a manner in which a goal can be satisfied, and may consist of a set of dependencies. Goals are a desired state of the system. Soft goals are goals that typically do not have a clear-cut satisfaction condition. Alternative tasks usually have different soft goals, which can introduce conflicts. The actors need to decide which soft goal to prefer and therefore which task to perform. The dependencies represent a service-based system, where one actor provides a specific service for other agents. The agents do not have any obligation to provide services, but may have internal beliefs that motivate them to provide those services. These beliefs are represented by either soft or hard goals. An example of a soft goal would be to keep the customer happy, where as a hard goal would be to never charge a customer for work that was not done. The *i** model is designed to show how different entities are dependant on services available by other agents in the system. The flow of information can also be visualized, by the *i** model. The *i** framework is designed to be used in early requirements elicitation. The framework guides how the system is designed, but it does not contribute to the specific implementation of the system. It also allows different strategic models to be designed in order to allow a system to behave in a differently.

2.1.6 *i** Summary

The *i** methodology works well in environments where the agents in the system are not considered to be benevolent. The strategic dependency model works well for specifying how resources can be shared, but the lack of formal modeling and the fact that *i** was developed for the early requirements phase makes it non-optimal for designing an entire system.

2.2 Agent-based Methodologies

The specification of multiagent systems is not a trivial task. A large amount of work has been devoted to making specification of multiagent systems a sound and complete process. The following section is devoted to work in this field.

2.2.1 Gaia

The Gaia methodology was the first widely publicized methodology for designing multiagent systems [30]. Gaia however was developed for a specific problem domain, closed multiagent systems with a static structure. The designers of Gaia realized that object oriented specification would not be able to properly represent the semantic meaning of a majority of multiagent systems. An agent-oriented abstraction has been used in virtually all other agent-based systems. Gaia also introduced a refinement process for eliciting sub-organizations shown in the analysis phase as shown in Figure 2. This process has been adopted from the software engineering discipline, where abstract entities (goals) are progressively refined into more concrete entities each time an iteration is applied.

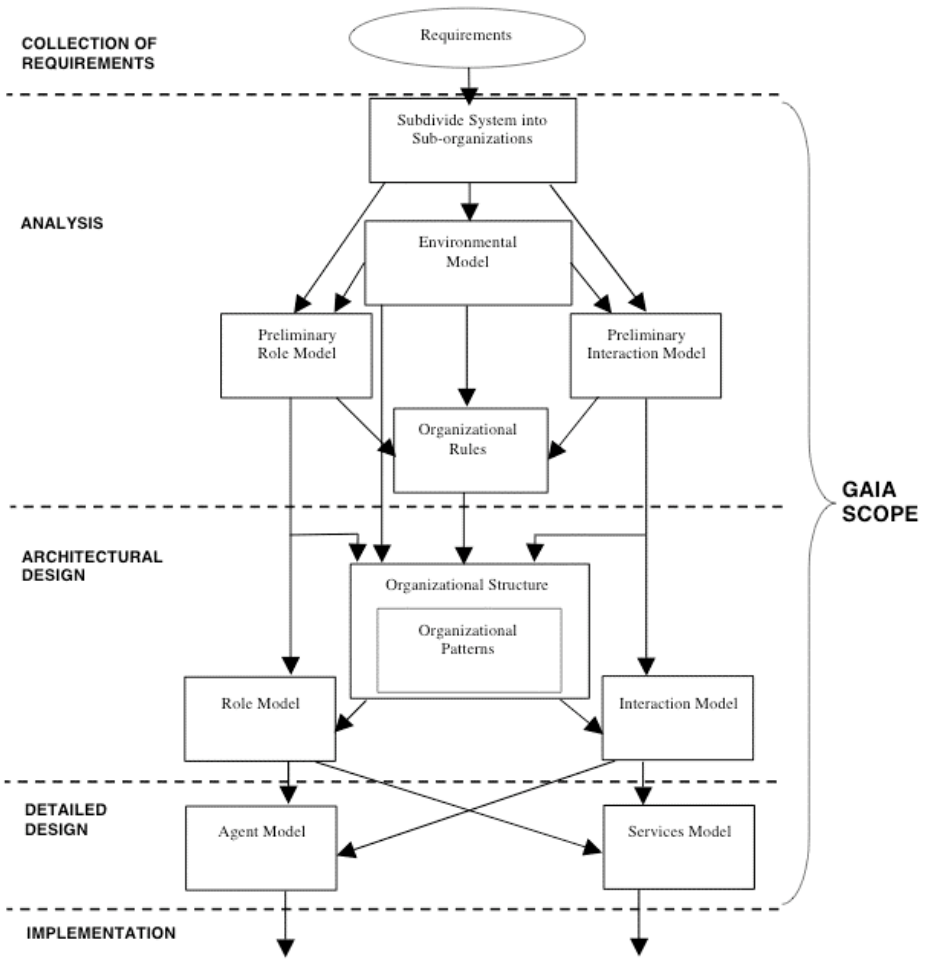


Figure 2. Gaia Framework **[33]**

Gaia also has the notion of and organization and a sub-organizations. A sub-organization is a decomposition at the organization level when the complexity of a single organization gets to large as Gaia does not scale very well. A sub-organization is a decomposition of the system. In Gaia the organizations are refined iteratively into smaller more concrete sub-organizations. This refinement was based on the Fusion methodology [8]. In Gaia there is no notion of goals, but in [33] they extend Gaia to elicit goals and sub-goals. The sub-goals are elicited through decomposition and then are attached to specific roles that are associated with those sub-goals. Gaia uses a notion of roles to model the responsibilities of agents in the system. A role is an abstract way to represent the manner in which an agent functions. Roles are key to Gaia because the roles handle the management of such things as protocols, resource responsibilities and what type of activities are allowed between agents. An example of the specification of a role is in

Code Listing 6. Roles are defined in a semiformal manner that maps roles to environmental resources.

```
reads   Papers  // allthepapersitreceives
changes   ReviewForms  // oneforeachofthepapers
```

Code Listing 6[33]

Agents are autonomous and therefore their function cannot be defined precisely. Agents can play a role or set of roles in system in order to provide the functionality of that role to the system. The main limitations of Gaia are the lack of requirements elicitation, an implied static structure, the ability to handle only a small number of agents, the closed nature of the system and static nature of agents' abilities. Dynamic multiagent systems cannot be specified very well within these design limitations. A dynamic system needs to be able to change the structure as the needs of the system change, and a static structure cannot handle this abstraction. A multiagent framework should also be able to handle large amounts of agents in order to be used in real world situations. The largest shortcoming however is the lack of support dynamic abilities of agents. A complete agent-based systems needs to be able to handle the dynamic abilities of agents. Agents cannot be assumed to have static abilities, when hardware can degrade, get better, or fail in the field. Because Gaia was a very general methodology for agent-based system design, Gaia has been extended by many other methodologies.

2.2.2 ROADMAP

The ROADMAP methodology is based on the Gaia methodology [19]. ROADMAP has been designed to take into account the dynamic nature of agent based systems. Gaia has trouble taking into account the dynamic nature of agent-based systems. ROADMAP allows the system to dynamically change the system behavior. This allows ROADMAP designed systems to be less rigid, and more dynamic than systems developed with Gaia alone. ROADMAP is designed for open-systems, where agents are heterogeneous and can freely join or leave the system. The behavior of the system is more dynamic than traditional Gaia model. This is done by using a roles hierarchy, which allows the agents to choose the best roles to achieve a goal. The ROADMAP designers noted that Gaia lacked the ability to do the following: elicit requirements, environmental information, domain knowledge model, non-hierarchical role model, and a lack of dynamic reasoning. The ROADMAP methodology tries to address all of these deficiencies by extending the Gaia methodology. ROADMAP adds to the specification phase the models for use-case diagrams, domain knowledge, environment knowledge and hieratical roles. ROADMAP also specifies that the diagrams should use the standard Agent Unified Modeling Language (AUML) [24] for modeling. AUML is designed for specifying agent-based system in UML using templates specific to agent-based systems. An example of a template for agent communication is in Figure 3. The agent/roles in the boxes and the communication between agents are denoted by CA-1 and CA-2.

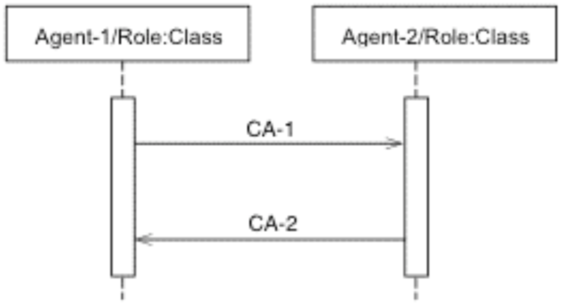


Figure 3. Roadmap Communication Template

These designs can include state charts, sequence diagrams, and protocol specifications. The designer will take a template (as in Figure 3) and add more specific details as needed. ROADMAP unlike Gaia uses aggregation in the role model. The role model is a tree, where the leaves of the tree are single roles (just as in Gaia) that cannot be decomposed any further. All the high-level roles are composed of 2 or more low-level roles. The low-level roles are aggregated into the high-level roles and their functionality is not overwritten by the high-level roles. ROADMAP does not provide any additional model checking properties to Gaia, but removes some constraints on what systems can be designed.

2.2.3 Tropos

The Tropos methodology was designed to bridge the gap between standard requirements analysis and the agent-based system analysis. Tropos uses the *i** framework to represent the entities. Tropos is designed for use in four separate design phases [23]. These phases are early requirements, late requirements, architectural design, and detailed architectural design.

As mentioned above eliciting requirements is a major process that must be done right. The early requirements are meant to elicit the *i** entities. These entities are the actors and goals of the system, along with the strategic dependency model. In the early requirements phase Tropos locates the means-ends relationships of the actors and the goals to determine the strategic dependency model (as in Figure 1**)**. In order to adequately represent the requirements, and address some shortcomings of *i**, Tropos adds a strategic rational model. This model provides more guidance on the manner in which the agents reason. This allows more reasoning to done on the model to better determine the behavior of the system. An example of the strategic rational model is in Figure 4.

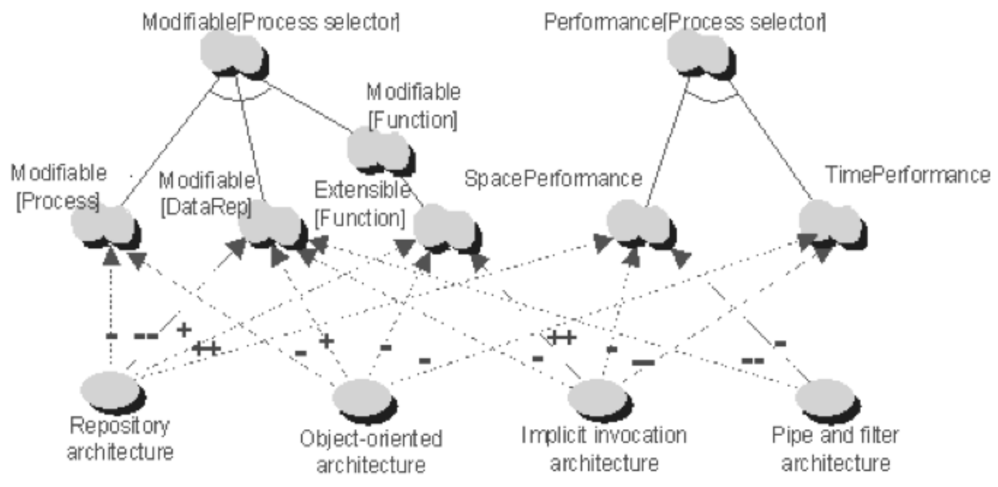

Modifiable[Process selector]
Performance[Process selector]
Modifiable [Function]
Modifiable [Process]
Modifiable [DataRep]
Extensible [Function]
SpacePerformance
TimePerformance
Repository architecture
Object-oriented architecture
Implicit invocation architecture
Pipe and filter architecture

Figure 4. Tropos Strategic Dependency Model **[23]**

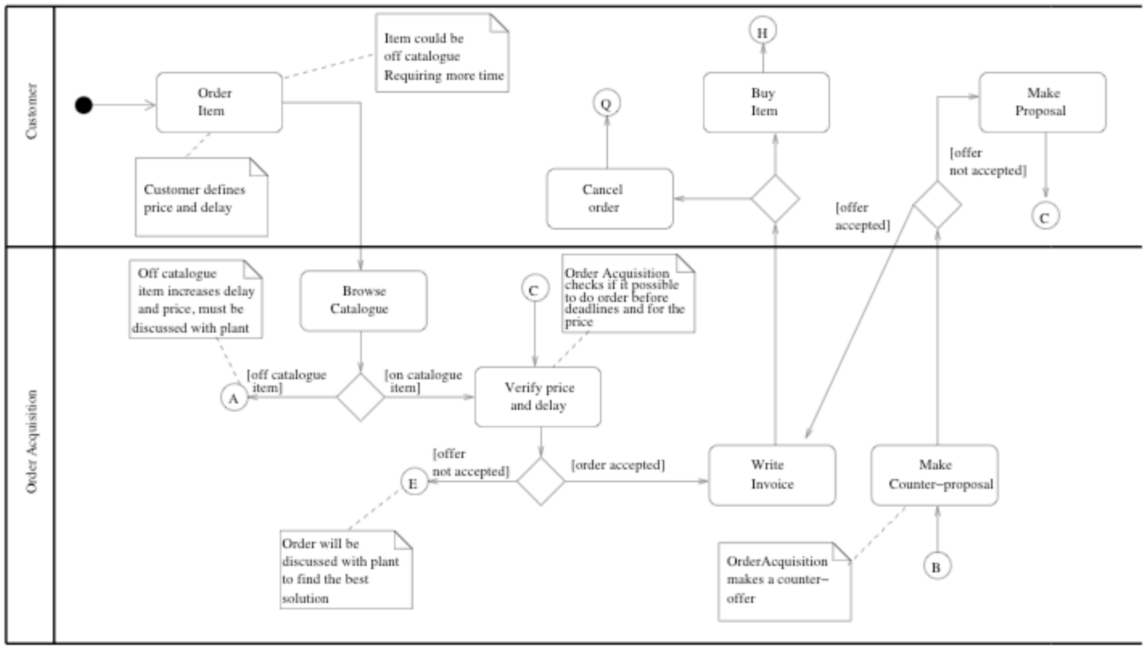
The late requirements provide documentation of all the functional and non-functional requirements of the system. The late requirements include the entities defined in the early requirements along with entities that may not be part of the system (non-functional). These requirements may decompose soft-goals into non-functional requirements and tasks. The non-functional requirements give insight into motivations of external actors.

The architectural design specifies how the entities are interconnected. A variety of architectures systems have been enumerated, each having benefits and drawbacks. Examples of architectures include Independent Events, Call-and-Return, Data Centered, and Virtual Machine. An analysis of the requirements provides incite into which architecture to choose for the various interconnections of the system. This phase also allows the designer to specify how a goal can contribute either positively or negatively to the completion of another goal (through the use of labels).

The detailed architectural design refines the aforementioned phase. This provides more fine-grained details of the interactions between actors in the system. These details may include but are not limited to protocols, languages, and messaging systems. These detailed designs typically are defined in AUML [24].

2.2.4 UML modeling Tropos

Huget created an extension [18] of Tropos that used UML 2 Activity diagrams to model goals and agents. This extension allows the goal model for a system to be more precisely defined, and thus more easily reasoned over. This extension uses the UML 2 diagrams to model the goals of the system. The UML 2 extension uses UML actions, activity edges, control nodes, and exception handling. In Figure 5 there are several goals (located in the rounded boxes) such as Order Item, Browse Catalog, and Buy Item. These goals are connected by activity edges (lines) and the activities connect to other goals or control nodes (diamonds). The goal model in Figure 5 has a clear meaning by just looking at the diagram.



Customer
Order Acquisition
Order Item
Item could be off catalogue Requiring more time
Customer defines price and delay
H
Q
Buy Item
Cancel order
Make Proposal
[offer not accepted]
[offer accepted]
C
Off catalogue item increases delay and price, must be discussed with plant
Browse Catalogue
C
Order Acquisition checks if it possible to do order before deadlines and for the price
A
[off catalogue item]
[on catalogue item]
Verify price and delay
[offer not accepted]
E
[order accepted]
Write Invoice
Make Counter–proposal
Order will be discussed with plant to find the best solution
OrderAcquisition makes a counter–offer
B

Figure 5. UML 2 Goal Model **[18]**

These models of the goals more completely define the semantics of the following i* entities: soft goals, goal contribution, goal constraints, events, resources, concurrency issues, priorities, and exceptions. The proposal also suggests that the system be divided into three specification sections (action level, plan level, and goal level). Each of these sections is specified more precisely than if it were specified in *i**. The use of UML 2 would also allow the reuse of tools developed for UML 2, and allow the specification to conform to a standard.

2.3 Applying goal modeling to MAS

The goal models listed above are good at doing many things, such as eliciting requirements, decomposition, and specifying goal interactions. Each model does one thing well, but none of the above methodologies is complete, and additional features would benefit goal modelers. An improved goal model (methodology) should include all of those items, and we propose that a goal model that uses goals in the implementation and at runtime would have added benefits. This model should introduce the ability to design goal models, verify properties of these models, and implement these models in the agents. This model should use an and/or goal decomposition, similar to the model used by KAOS. The goal model should include the ability to specify additional characteristics of goals. Examples of these additional characteristics are the ability to create/destroy goals. A trigger allows the model to become much more dynamic and adaptable. The model must also allow for ordering of goals via goal precedence. This precedence allows the system designer to specify a partial linear ordering of goal achievement.

Chapter 3 GMoDS Definition

This chapter defines the Goal Model for Dynamic Systems (GMoDS), which includes triggering and precedence. In Section 3.1 the GMoDS is formally defined. Section 3.2 describes the Goal Specification Tree. Section 3.3 describes goal triggering, and Section 3.4 describes goal precedence. Section 3.5 describes the Goal Instance Tree and how it relates to the Specification Tree. The definitions lay the groundwork for the Execution Model in Chapter 4.

3.1 GMoDS Definition

The GMoDS specification model includes three main entities: goals, events, and parameters. A goal (Goal) is an observable desired state of the world and an event (Event) is an observable phenomenon that occurs in the environment. Parameters provide a specification of the inputs expected for a goal from some event that occurs. Runtime Parameters provide runtime information to agents on how a goal is achieved. This information allows Goals and Events to be defined generically and unique instances to be created during runtime. In order to define a Goal and an Event properly, parameters (Parameters) must also be formally defined first. Formal parameters (FormalParameters) are a set of parameters that are used in specification. A parameter type (ParameterType) defines the name and type of a specific parameter. The actual parameters (ActualParameters) consist of a set of parameter types (ParameterType) with a set of associated values. The values of ActualParameters are determined at run-time.

ParameterType	⟨name, type⟩	**D1**
FormalParameter	⟨Set(ParameterType)⟩	**D2**
ActualParameter	⟨FormalParameters, Set(values)⟩	**D3**
Parameters	⟨name,type⟩ where type = ⟨FormalParameter|ActualParameter⟩	**D4**
Event	⟨name,Parameters⟩	**D5**
Goal	⟨name,Parameters⟩	**D6**

While an agent is performing a goal, a number of events may occur. These events may cause other goals in the system to be added or removed. The new goals added to the system are parameterized based on the Parameters of the event that caused them. The goals that are removed from the system are removed based on the parameters that were used to create them. Therefore, we define an Event to have a name and a set of Parameters. An agent specializes its performance of the roles that it plays based on the Parameters of the associated goal. A Goal is thus defined to have a name and Parameters.

The above definitions allow the formal definition of GMoDS through the use of a set of predicates, relations, and functions. A goal has two specializations as in Figure 6. The first are *Goal Classes* which are goals specified by the system designer to model the types of goals within the organization. The specification of these goal classes allow the designer to model relationships and attributes of the goal abstraction. Goal classes are restricted to having only FormalParameters as Parameters. The second are *Goal Instances*, which represent an instantiation of a specification goal. The instance/instanceOf relations represent the relation between goal classes and goal instances. As denoted by the diagram each goal instance is an instance of exactly one goal class, and a goal class may have zero or more goal instances. Goal instances have ActualParameters as Parameters The set of ParameterTypes of the goal class must match the set of ParameterTypes in the goal instance. This ensures that the goal instance is indeed a proper instance of the goal class it is implementing. This axiom is provided in the section after the Goal Instance tree has been further defined in Section 3.2.2 constraint C7.

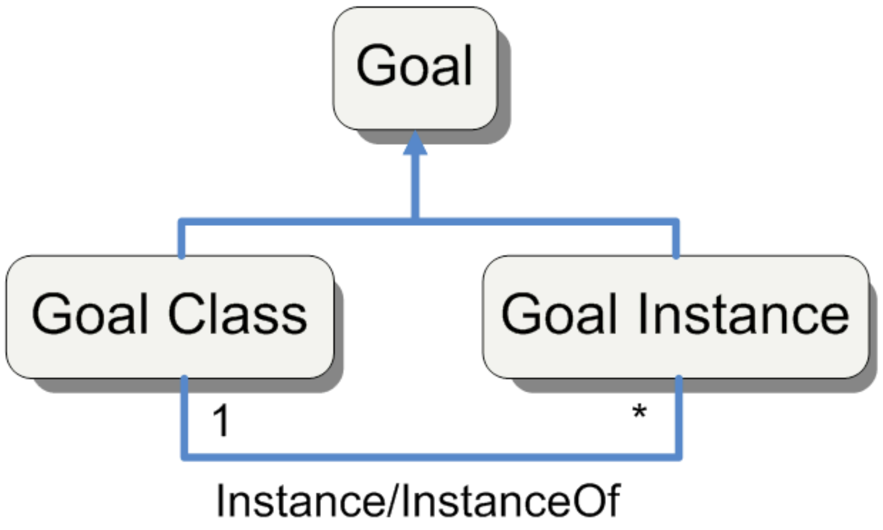


Figure 6. Goal Specializations

The system designer specifies a finite number of specification goals and, therefore a set that includes all the specification goals can be defined (G_S). Each event in the system is defined as an entity, and can be referenced by a name. The system designer specifies a finite number of events that the system can recognize. Because this set is finite, a set of specification events can be defined (E_S). Every event in E_S has Parameters, and those parameters have to be of the type FormalParameters. This restriction is defined in Section 3.2.2.

G_S	Set(Goal)	**D7**
E_S	Set(Event)	**D8**

3.2 Goal Specification Tree

Defining goal trees using GMODS is a natural extension of the classic top down approach of problem decomposition. Goal trees give a hierarchical structure for the set of goals in the system. An example of such a goal tree is shown in Figure 7. Upper level goals are decomposed into lower level goals. The upper level goals are known as parent goals, and the lower level goals are known as child goals. Goal trees require the parent goals to specify an achievement condition. A parent goal can have either a conjunctive or disjunctive achievement condition. A conjunctive parent goal (denoted by <<and>>) is achieved when all of the children are achieved. A disjunctive parent goal (denoted by <<or>>) is achieved when at least one of its children is achieved. Parent goals without any children are known as leaf goals.

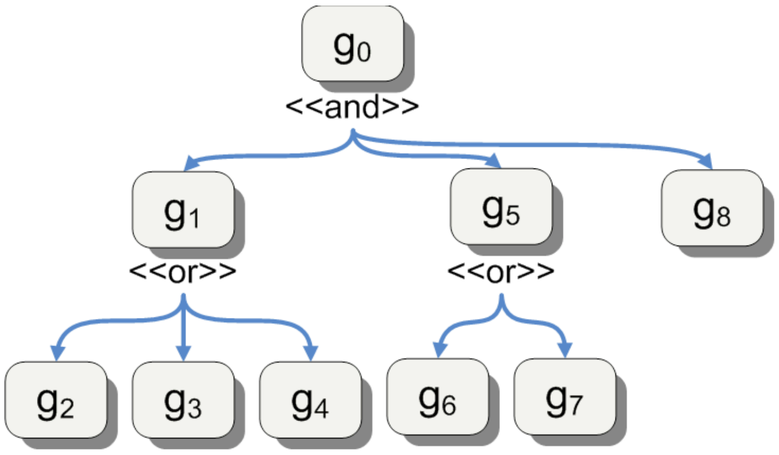


Figure 7. Example Goal Model

A GMoDS specification tree (G_S) specifies how the goal classes are related to one another. The goal classes in the goal specification tree are analogous to the specification of classes in an object-oriented language. Classes are designed before hand, and then are dynamically instantiated at runtime, with each instance having its own set of attributes. The set of attributes is based on the class definitions that were elicited during the requirements phase. The runtime instances run independently of each other, but are based on the same template. The notion of instances can also be applied to the goal trees. A specification goal tree also has an associated instance tree that is created at runtime. The Parameters of the instance goals (ActualParameters) are based on the

Parameters given in the specification goals (FormalParameters). The goal instances are allocated in the Goal Instance Tree (G_I) during runtime.

Each goal instance is achieved independently of every other instance of that goal. For example, suppose that goal g_5 in Figure 7 is to rescue a victim and there are three instances of g_3. Then, there would be three victims to be rescued and the achievement of each one of these rescue goals is independent of the achievement of every other rescue goal.

3.2.1 Goal Classes

Every goal class in G_S is described by the following predicates: root, conjunctive and disjunctive. Goal classes also have a preference function that maps to real values from 0…1.

Root, *preference*, *conjunctive*, and *disjunctive* are statically defined in the specification tree. The *conjunctive* and *disjunctive* predicates are defined for all goal classes. Continuing with the example in Figure 7 conjunctive(g_0) = False and disjunctive(g_0) = True. The *root* predicate is True for only the single root of the tree. Therefore in Figure 7 root(g_0) = True and root(g_3) = False. The standard convention is to label the root goal class as g_0.

The *preference* function allows the system designer to specify which disjunctive goals are preferred when the system is faced with a choice. Suppose that there is a set of disjunctive goals and there are not enough agents in the system to work simultaneously on every goal in the set. The runtime reasoning can then attempt the assignments for the subset of goals with the highest preferences.

root :	$G_S \rightarrow$ Boolean	**D9**
conjunctive :	$G_S \rightarrow$ Boolean	**D10**
disjunctive :	$G_S \rightarrow$ Boolean	**D11**
preference :	$G_S \rightarrow 0\ldots1$	**D12**

Along with entities in the system, there is a set of relations that defines how the different entities interact with one another. The subgoal and precedes (Section 3.4) relate two goals to one another. The triggers and ¬triggers relations (Section 3.3) relate two goals and an event to one another.

subgoal	$\subseteq G_S \times G_S$	**D13**
precedes	$\subseteq G_S \times G_S$	**D14**
triggers	$\subseteq G_S \times E_S \times G_S$	**D15**
¬triggers	$\subseteq G_S \times E_S \times G_S$	**D16**

Defining a set of functions for G_S allows the application of semantic meaning and logical reasoning to be given to the tree. These functions define the relationships among goals. Goals in G_S can only be mapped to other goals in G_S and events in E_S. The subgoal relation defines how the high-level goal classes in G_S are decomposed into lower-level goals. The parent and child mappings are derived from the subgoal relation.

children : $G_S \rightarrow P(G_S)$ **D17**

parent : $G_S \rightarrow P(G_S)$ **D18**

The definition of the parent and child functions can also be formally stated. The function definition selects all goal pairs that are elements of the children relation.

parent(g) $= \{g_1{:}G_S \mid (g_1,g) \in \text{subgoal}\}$ **D19**

children(g) $= \{g_1{:}G_S \mid (g,g_1) \in \text{subgoal}\}$ **D20**

For example in Figure 7 children (g_0) = $\{g_1,g_5\}$, and parent(g_3) = $\{g_1\}$. The parent function is applied in the reverse direction of the children function, and vise-versa. The parent and children functions allow for single relationships to be queried. The more interesting cases come with the definition of the transitive closures of those relations. The transitive closure, combined with the triggers and precedes relations, are important when defining an advanced runtime reasoning algorithm, and the function signatures are defined below.

children$^+$: $G_S \rightarrow P(G_S)$ **D21**

parent$^+$: $G_S \rightarrow P(G_S)$ **D22**

The transitive closure of the children relation includes all the goals in the children relation, along with each of the childrens' children, until no more children can be added. The transitive closure of the parent relation includes all the parent goals and those parents' parents, until the root goal is added to the set. The formal definitions of these functions are stated below.

children$^+$(g) $= \{g_1{:}G_S \mid (g,g_1) \in \text{subgoal} \vee [(g_2,g_1) \in \text{subgoal} \wedge g_2 \in \text{children}^+(g)]\}$ **D23**

parent$^+$(g) $= \{g_1{:}G_S \mid (g_1,g) \in \text{subgoal} \vee [(g_1,g_2) \in \text{subgoal} \wedge g_2 \in \text{parent}^+(g)]\}$ **D24**

3.2.2 G_S Restrictions

In G_S, goals that have children (high-level goals) must either be conjunctive or disjunctive. Allowing a goal to be both conjunctive and disjunctive would lead to ambiguity about the satisfaction condition of the goal. Therefore we define the following axiom (exclusive or denoted by $\otimes$).

$$\forall\, g : G_S \mid children(g) \neq \{\} \;\Rightarrow\; conjunctive(g) \otimes disjunctive(g) \qquad \textbf{C1}$$

Instances of leaf goals are the goals that are actually assigned to agents. The leaf goals in Figure 7 are $\{g_2,g_3,g_4,g_6,g_7\}$. Because leaf goals are directly achievable they are neither conjunctive nor disjunctive, as they are either achieved or not. The following axiom defines the invariant formally.

$$\forall\, g : G_S \mid children(g) = \{\} \Rightarrow \neg(conjunctive(g) \vee disjunctive(g)) \qquad \textbf{C2}$$

The goal specification tree is required to follow the standard conventions of a tree. The definition of a tree includes the invariant that there are no cycles. A goal that is an element of the transitive closure of the children relationship is a member of a cycle. The following axiom provides this restriction formally.

$$\forall\, g : G_S \mid g \notin children(g)^{+} \qquad \textbf{C3}$$

The definition of any tree also requires there to be a single root goal. The root goal is the ancestor of every other goal in the tree. The function root returns true only for the root of the tree, and returns false for every goal that is not the root.

$$\#\{g \mid g \in G_S \wedge root(g)\} = 1 \qquad \textbf{C4}$$

In Figure 7 $root(g_0)$ = True and $root(g_3)$ = False. The root goal also does not have a parent, and every other goal in the goal specification tree must have just a single parent goal. The following axioms state this formally (cardinality is denoted by # operator).

$$\forall\, g: G_S \mid \neg root(g) \Rightarrow \#parent(g) = 1 \qquad \textbf{C5}$$

$$\forall\, g : G_S \mid root(g) \Rightarrow \#parent(g) = 0 \qquad \textbf{C6}$$

This axiom combined with the restriction that there are no cycles in the goal tree ensures that there is a single root of the goal tree.

Events in E_S are required to have only formal parameters. This is stated formally by the following axiom.

$$\forall\, e : E_S \mid e.Parameters.type = FormalParameters \qquad \textbf{C7}$$

3.3 Goal Triggers

Goal instances are created based on the occurrence of specific events in the system or the environment. As events in the system occur, the goal tree needs to dynamically adjust to those events. Creating and destroying goal instances provides this adjustment. To continue with the parallel with object-oriented software, the goal instances are similar to object instances. Goal instances are dynamically created and destroyed. The *triggers* relation creates a new parameterized instance goal based on the goal class. The events labeled as *negative triggers* (abbreviated by ¬triggers) allow goals to be removed from the system.

The *triggers* and *¬triggers* relations map one specification goal to another via a parameterized event. The triggers relations allow one goal instance to be created/removed by a second goal instance when a specific event occurs. The triggers/¬triggers functions gather all the goals that are triggered by a single goal in the specification tree.

$$\text{triggers} : G_S \rightarrow P(G_S) \qquad \textbf{D25}$$

$$\neg\text{triggers} : G_S \rightarrow P(G_S) \qquad \textbf{D26}$$

These functions are defined as follows.

$$\text{triggers}(g) = \{g_1{:}G_S \mid \exists\ e{:}E_S\ (g,e,g1) \in \text{triggers}\} \qquad \textbf{D27}$$

$$\neg\text{triggers}(g) = \{g_1{:}G_S \mid \exists\ e{:}E_S\ (g,e,g1) \in \neg\text{triggers}\} \qquad \textbf{D28}$$

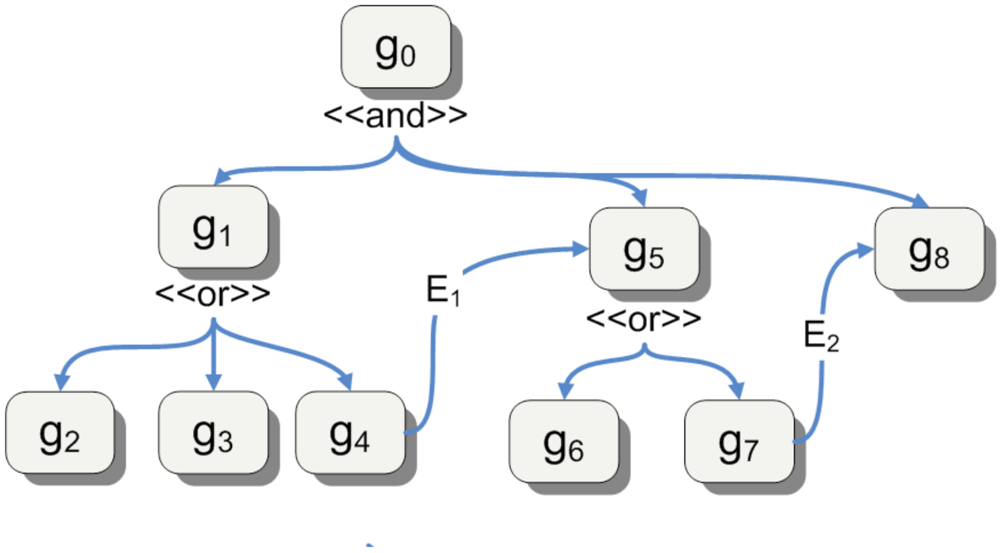


Figure 8. Goal Model With Events

Figure 8 refines Figure 7, by adding a trigger relation between from g_4 to g_5 and from g_7 to g_8. An example of the use of the triggers function would be triggers(g_4) = {g_5} and triggers(g_7) = {g_8}. The triggers relation would include the tuples (g_4, e_1, g_5) and (g_7, e_2, g_8).

To properly understand the impact of the triggers relation, it is vital to understand the consequences of a goal being triggered. The triggers closure is more inclusive than the transitive closure of the triggers relation, as the children of a triggered goal can trigger additional goals in the tree. Therefore we define the triggers closure to accommodate to this situation. The triggers closure collects the entire set of goals that can be eventually triggered by some goal. This closure must include all goals that can be directly triggered or goals that are triggered indirectly. In the example in Figure 8, the set of goals that g_4 directly triggers is $\{g_5\}$. The goals that are indirectly triggered are $\{g_6,g_7,g_8,g_9\}$. Therefore the value of the triggers$^+$ relation of goal g_4 is $\{g_5,g_6,g_7,g_8,g_9\}$. The triggers closure of the ¬triggers uses the same process as the triggers closure (triggers$^+$). The triggers$^+$ (D29) and ¬triggers$^+$ (D30) relations are formally be defined by the two following set of recursive axioms.

$$\begin{aligned}
&\forall\, g,g_1,g_2 : G_S \\
&(g,g_1) \in \text{triggers} && \Rightarrow (g,g_1) \in \text{triggers}^+ \\
&(g,g_1) \in \text{triggers}^+ \wedge (g_1,g_2) \in \text{subgoal} && \Rightarrow (g,g_2) \in \text{triggers}^+ \\
&(g,g_1) \in \text{triggers}^+ \wedge (g_1,g_2) \in \text{triggers} && \Rightarrow (g,g_2) \in \text{triggers}^+
\end{aligned}$$

D29

$$\begin{aligned}
&\forall\, g,g_1,g_2 : G_S \\
&(g,g_1) \in \neg\text{triggers} && \Rightarrow (g,g_1) \in \neg\text{triggers}^+ \\
&(g,g_1) \in \neg\text{triggers}^+ \wedge (g_2,g_1) \in \text{subgoal} && \Rightarrow (g,g_2) \in \neg\text{triggers}^+
\end{aligned}$$

D30

The triggers$^+$ and ¬triggers$^+$ are functions that map a goal to a subset of the goals in G_S based on the triggers$^+$ and ¬triggers$^+$ relations. These functions are formally defined below.

$$\text{triggers}^+ : G_S \rightarrow P(G_S)$$ **D31**

$$\neg\text{triggers}^+ : G_S \rightarrow P(G_S)$$ **D32**

$$\text{triggers}^+(g) = \{g_1{:}G_S \mid g,g_1 \in \text{triggers}^+\}$$ **D33**

$$\neg\text{triggers}^+(g) = \{g_1{:}G_S \mid g,g_1 \in \neg\text{triggers}^+\}$$ **D34**

The reverse mapping of the triggers and ¬triggers functions can also be defined. The reverse function gathers all the goals than can trigger a particular goal on any event. The definition of these functions gather the set of goals that triggers a particular goal. These functions will gather all goals that are triggeredBy or ¬triggeredBy over all events.

$$\text{triggeredBy} : G_S \rightarrow P(G_S)$$ **D35**

$$\neg\text{triggeredBy} : G_S \rightarrow P(G_S)$$ **D36**

These functions map a goal to a set of goals. The functions are formally defined over the triggers and ¬triggers relations.

$$\text{triggeredBy}(g) = \{g_1{:}G_S \mid \forall\, e{:}E_S\ (g_1,e,g) \in \text{triggers}\}$$ **D37**

$$\neg\text{triggeredBy}(g) = \{g_1{:}G_S \mid \forall e{:}E_S\ (g_1,e,g) \in \neg\text{triggers}\}$$ **D38**

The triggeredBy^+ and $\neg\text{triggeredBy}^+$ functions can also be defined for the reverse mapping of the triggeredBy relations. These functions map a goal to a set of goals that can (positively or negatively) trigger that goal.

$$\text{triggeredBy}^+ : G_S \rightarrow P(G_S) \qquad \textbf{D39}$$

$$\neg\text{triggeredBy}^+ : G_S \rightarrow P(G_S) \qquad \textbf{D40}$$

These functions are defined using the triggers closures to find all goals that include the goal in the domain in the triggers closures.

$$\text{triggeredBy}^+(g) = \{g_1 : G_S \mid g \in \text{triggers}(g_1)^+\} \qquad \textbf{D41}$$

$$\neg\text{triggeredBy}^+(g) = \{g_1 : G_S \mid g \in \neg\text{triggers}(g_1)^+\} \qquad \textbf{D42}$$

Based on the definition of the parent relation and goal triggering, the triggered parent relations can be formally defined. These functions are used in the reasoning algorithm in G_I, as defined in Section 3.5.

$$\text{triggeredParents} \quad G_S \rightarrow P(\ G_S) \qquad \textbf{D43}$$

$$\text{leastTriggeredParent} \quad G_S \rightarrow G_S \qquad \textbf{D44}$$

$$\text{LCTP} \quad (G_S\ , G_S) \rightarrow G_S \qquad \textbf{D45}$$

The triggered parents function gathers all the goals in the transitive closure of the parent relation that do not have an empty triggeredBy relation.

$$\text{triggeredParents}(g) = \{g_1 : G_S \ \mid g_1 \in \text{partents}^+(g) \wedge \text{triggeredBy}(g_1) \neq \{\}\} \qquad \textbf{D46}$$

Figure 8 examples would be $\text{triggeredParents}(g_6) = \{g_0, g_5\}$ and $\text{triggeredParents}(g_2) = \{g_0, g_1\}$. The goals g_0 and g_1 may appear at first, to not belong, but they have the implicit initial trigger, which is defined in Section 3.3.2.

The leastTriggeredParent relation finds the lowest parent goal that is triggered. This lowest goal is the goal that is a triggered parent that includes all other triggered parents in the transitive closure of the parent relation.

$$\text{leastTriggeredParent}(g) = \{g_1 : \text{TriggeredParents}(g) \mid g_2 \in \text{TriggeredParents}(g) - g_1 \Rightarrow g_2 \in \text{parent}^+(g_1)\ \} \qquad \textbf{D47}$$

In the example from Figure 8 $\text{leastTriggeredParent}(g_6) = \{g_5\}$ and $\text{leastTriggeredParent}(g_2) = \{g_1\}$. The Least Common Triggered Parent (LCTP) takes two goals and finds the lowest parent goal that is common to both of those goals and that common goal is also triggered. To find this goal the intersection of the TriggeredParents is found. This yields all the goals that are in common to both goals. Then the goal that is a child of all other goals in the intersection is located.

$$LCTP(g_1,g_2) = \{g_3 : TriggeredParents(g_1) \cap TriggeredParents(g_2) \mid g_4 \in [TriggeredParents(g_1) \cap TriggeredParents(g_2)] - g_3 \wedge g_4 \in parent^+(g_3)\}$$ **D48**

The example in Figure 8 $LCTP(g_2,g_6) = g_0$.

3.3.1 Trigger Restrictions

There is a restriction on the triggers relations that pertain to how the designer can design the goal specification tree. This restriction has been put in place to remove ambiguity that can occur. Suppose the designer had placed a trigger between g_4 and g_6 as shown in Figure 9.

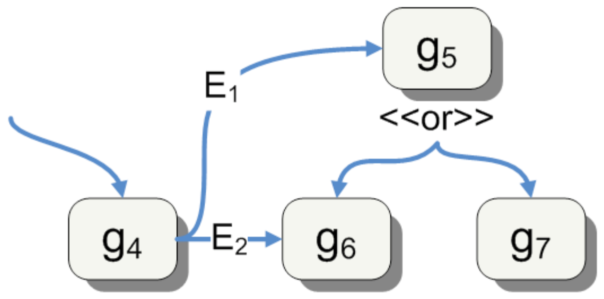


Figure 9. Illegal Triggers

If event e_2 were to occur before any event e_1 were to occur, there would exist no instance of g_5, and therefore g_6 would have no parent goal. The situation where a parent goal is a triggered goal, and a child goal is triggered by a goal that is not a member of the children the of parent goal is ambiguous, and therefore disallowed. We therefore define an axiom to disallow this situation. The ambiguity is disallowed by ensuring that all inbound triggers relations come from the children of the triggered goal. It is stated formally by the following axiom.

$$\forall g_1,g_2,g_3{:}G_S \mid g_2 \in triggers(g_1) \wedge g_3 \in children^+(g_2) \Rightarrow triggeredBy(g_3) \subseteq children^+(g_2)$$ **C8**

3.3.2 Initial Goal Trigger

Goals from G_S are added to G_I by events defined in E_S occurring in the system. For a system to start, there must exist an event that adds goals (including the root goal) to G_I. However if there are no goals in G_I then events cannot occur (classic bootstrapping problem). To solve this problem we add an *initial trigger* to the root goal. When the system starts, the initial event occurs and the root goal is added to G_I. The algorithm then systematically and recursively adds all children to G_I that are not triggered by some other event in E_S.

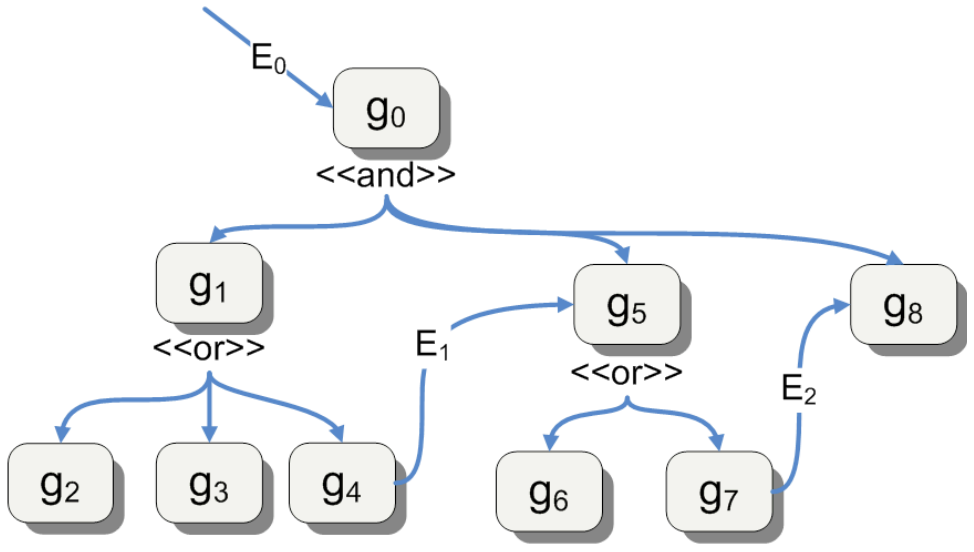


Figure 10. Goal Model With Initial Triggers

The initial event annotation has been added in Figure 10 (denoted by e_0). When e_0 occurs the goals that are added to G_I are as follows. Step 1 simply adds g_0 to G_I ($G_I = g_0$). Step 2 would add g_1 to G_I ($G_I = \{g_0,g_1\}$). Goals g_5 and g_8 are not added, as they are triggered by other goals triggeredBy(g_5) = $\{g_4\}$ and triggeredBy(g_8) = $\{g_7\}$. Step 3 would add g_2, g_3, g_4 to G_I ($G_I = \{g_0,g_1, g_2,g_3,g_4\}$).

3.4 Goal Precedence

To allow a full or partial ordered execution of goals in the system, the goal specification tree also allows a goal to precede a set of goals in the goal tree. Precedence ensures that no agents work on a specific goal until all goals that precede that goal have been achieved. An example of goal precedence would be that of object identification and then manipulation. The object must be first identified and then it may be manipulated. If the object is manipulated before it is identified then that manipulation may be improper. Object identification therefore precedes object manipulation. Full temporal goal precedence includes all instances of the preceded goals over time. Partial temporal goal precedence is over a set of goals in a specific sub-tree. The context of the precedence defines whether the ordering is full or partial. Goals that have a least triggered parent of the root goal g_0, have full precedence. Goals that have a least triggered parent less than the root goal have precedence over the goals that share a least common triggered parent. The *precedes* function is one that maps a goal in G_S to a set of goals from G_S.

$$\text{precedes: } G_S \rightarrow P(G_S) \qquad \textbf{D49}$$

The definition of the *precedes* function gathers all the goals that are in the precedes relation. The function definition is as follows.

$$\text{precedes}(g) = \{ g_1{:}G_S \mid (g,g_1) \in \text{precedes} \} \qquad \textbf{D50}$$

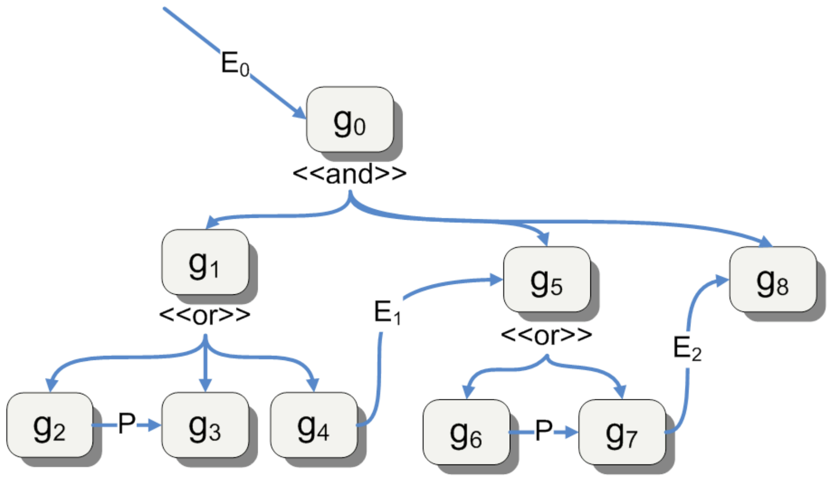


Figure 11. Goal Model With Precedence

Figure 11 is an extension of the example in Figure 10 with two precedence relations inserted. In this example precedes(g_2) = {g_3} and precedes(g_6) = {g_7}. *Full temporal goal precedence* implies that the preceding goal (g_2 or g_6) must be complete before any of the preceded goals (g_3 or g_7 respectively) can be worked on. *Partial temporal goal precedence* implies that only the preceding goals in the same subtree must be complete. Therefore, the context of the precedence determines the type of precedence. Goals that have a least triggered parent of the root goal g_0, have full precedence. Goals that have a least triggered parent less than the root goal have precedence over the goals that share a least common triggered parent. The leastTriggeredParent(g_2) = {g_0} and root(g_0) = True; therefore g_2 has full order precedence. The precedence relation between goals g_2 and g_3 means that in order for g_3 to be pursued, g_2 must be achieved. The semantics of the precedes relation between g_6 and g_7 is a partial order precedence.

Because g_5 (the lowest triggered parent) is triggered by event (e_1), the precedence between g_6 and g_7 is based only on a specific instance tree of g_5 on e_1. Therefore an instance of g_7 is preceded by the instance of g_6 that has the same parent in the instance tree (lowest triggered parent). The precise The reverse of the precedes relation can also be defined. The reverse function gathers all goals that precede the goal in the domain. In the example goal tree in Figure 11, precededBy(g_3) = {g_2} and precededBy(g_7) = {g_6}. This function also maps a goal to a set of goals, and is formally defined as follows.

$$\text{precededBy} : G_S \rightarrow P(G_S) \qquad \textbf{D51}$$

The precededBy function uses the precedes function to select the set of goals which include the domain goal in the precedes function. This is formally stated below.

$$\text{precededBy}(g) = \{g_1{:}G_S \mid \text{precedes}(g_1,g)\} \qquad \textbf{D52}$$

The precedence closure can also be defined in a manner similar to the triggers closure. The precedence closure includes all the goals that are preceded by the domain goal, all of the children of goals that in preceded closure, and all goals that are preceded by goals in the preceded closure. The process is also repeated for all goals in the relation, until the set does not change. The formal definition of the precedence closure relation is defined below.

$$g,g_1,g_2 : G_S$$

$$\begin{array}{ll}
\{g,g_1\} \in \text{precedes} & \rightarrow (g,g_1) \in \text{precedes}^+ \\
\{g,g_1\} \in \text{precedes}^+ \wedge \{g_1,g_2\} \in \text{subgoal} & \rightarrow (g,g_2) \in \text{precedes}^+ \\
\{g,g_1\} \in \text{precedes}^+ \wedge \{g_1,g_2\} \in \text{precedes} & \rightarrow (g,g_2) \in \text{precedes}^+
\end{array} \qquad \textbf{D53}$$

The function for the precedence closure of the precedence relation can be also defined. This function maps a goal to a set of goals.

$$\text{precedes}^+ : G_S \rightarrow P(G_S) \qquad \textbf{D54}$$

The definition of the precedes$^+$ function selects all the goals that are in the precedes$^+$ relation.

$$\text{precedes}^+(g) = \{\forall\, g_1{:}\, G_S \mid \{g,g_1\} \in \text{precedes}^+ \} \qquad \textbf{D55}$$

The precededBy closure function is the reverse mapping of the precedence closure. The function maps a goal to a set of goals.

$$\text{precededBy}^+ : G_S \rightarrow P(G_S) \qquad \textbf{D56}$$

The definition of the function includes all goals that include the domain goal in their precedence closure.

$$\text{precededBy}^+(g) = \{g_1{:}\, G_S \mid g \in \text{precedes}^+(g_1)\} \qquad \textbf{D57}$$

3.4.1 Goal Precedence Restrictions

There are several goal precedence restrictions on the goals in G_S. The first restriction is that of goal cycles. We can clearly see that if there exits a cycle in the precedes relationship, then none of the goals in the cycle will ever be assigned to any agents, and we must therefore disallow this situation. We define this restriction formally as follows:

$$\forall g : G_S \mid g \notin precedes^{+}(g) \qquad \textbf{C9}$$

It can also be observed that a goal must not precede any of its ancestors. If a goal were to precede a parent, it would imply that the child must be achieved before the parent could be pursued; however, because a parent is implicitly being pursued if one of its children is being pursued, this precedence cannot be valid. This restriction can be stated formally as follows.

$$\forall g : G_S \mid precedes^{+}(g) \cap parent^{+}(g) = \{\} \qquad \textbf{C10}$$

In addition to the pure precedence cycles, a cycle of mixed triggers and precedes relationships is disallowed. The mixed cycle creates a set of goals that cannot be assigned. For example suppose that goal g_1 is preceded by g_2, and g_1 triggers g_3, and g_3 precedes g_2 (depicted in Figure 12).

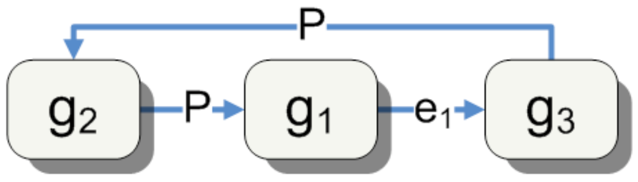


Figure 12. Illegal Precedence and Triggers

It can be observed that g_1 can never be assigned to any agent as a result of the precedes/triggers cycle. By the definition of precedence all triggered instances of g_3 must be complete, and no more instances of g_2 can occur for g_1 to assigned. However g_1 can trigger more instances of g_3, which precedes g_2. A restriction makes sure that all of the goals in the transitive closure of the triggers relation, do not include the goal in the domain in the transitive closure of the precedes relation. The restriction on the goal tree is stated formally below.

$$\forall g_1, g_2 \mid g_1 \in precededBy^{+}(g_2) \Rightarrow g_2 \notin triggers+(g_1) \qquad \textbf{C11}$$

3.5 Goal Instance Tree

3.5.1 Goal Instance Entity Definition

During the execution of the system, a variety of events can occur. Many events may occur during the runtime of the system. The GMoDS instance tree is only concerned with events that have been defined in E_S. An event can occur at any time during the runtime of the system while goals are assigned to agents. For reasoning purposes, a unique identifier attribute is added to the definition of the event instances in (E_I). This attribute is given a unique value during the runtime of the system, which allows the specific goal instances to be identified, as there could be events that have the same parameters and values.

$$\text{uid: } \quad \text{Event}_I \rightarrow 0 \ldots \infty \qquad \textbf{D58}$$

The goal instance tree contains of a single type of entity named goal instance (GoalInstance). Each goal instance has a name, a set of Parameters, and its uid. Each of these parameters has a value, which are set at runtime when the events occur. The parameter types and name are determined by the event in E_S, upon which the event is based. The goal instance definition is as follows.

$$\text{GoalInstance } \langle \text{name,Parameters,uid} \rangle \qquad \textbf{D59}$$

The definition of the *goal instance tree* (G_I) follows as a set of goal instances.

$$G_I : \text{Set(GoalInstance)} \qquad \textbf{D60}$$

The definition of G_I needs to be linked back to G_S for reasoning purposes. Using the object-oriented analogy, the link back to the class definition uses a function named *instanceOf* which maps an instance goal to its goal class.

$$\text{instanceOf } \subseteq G_I \times G_S \qquad \textbf{D61}$$

The instanceOf function maps a (goal class, goal instance) pair to a Boolean value. This function allows the relation between the G_I and G_S to be queried.

$$\text{instanceOf: } G_I \times G_S \rightarrow \text{Boolean} \qquad \textbf{D62}$$

The value is True if the goals are part of the instanceOf relation. This is formally defined below.

$$\text{instanceOf}(g_1,g_2) = (g_1,g_2) \in \text{instanceOf} \qquad \textbf{D63}$$

The instanceOf relation is set at the creation of a new goal instance. To ensure a goal instance can only be an instance of one goal class we can define the following restriction.

$$\forall\, g_1 \ : G_I\ g_2, g_3 \ : G_S \mid \text{instanceOf}(g_1,g_2) \wedge \text{instanceOf}(g_1,g_3) \rightarrow g_2 = g_3 \qquad \textbf{D64}$$

In the definition of the relationship between the goal specification tree and goal instances, it was stated that the goal instances must have the same parameter set as the goal instance from which they were derive. This can be stated by the following restriction.

$$\forall\, g_1 \in G_I\,,\ \exists\, g_2 \in G_S \mid \text{instanceOf}(g_1,g_2) \qquad \textbf{D65}$$
$$\Rightarrow g_1.\text{ActualParameters.FormalParemeters} = g_2.\text{FormalParameters}$$

3.5.2 Goal Instance relations

G_S allows the user to specify the system hierarchy and the relations between goal classes. The specification tree is used as a template by the system at runtime. The system can dynamically create goals in G_I. The G_I tree retains the static tree structure but still allows for dynamism by way of goal triggering. G_I is defined as the set of instance goals that have been created by the runtime system. The entity definition of the goal instance ensures that a goal instance is an instance of a single goal class. The goals in G_I retain the relationships that exist in G_S (subgoal, parent, children…) through the relations defined in G_S and the instanceOf relation. In addition to these relationships, an additional set of functions is defined over G_I.

The predicates *obviated*, *preceded*, *failed*, and *achieved* are all dynamically set in G_I. The *achieved* predicate states whether a goal has been achieved by the system. For leaf goals, *achieved* becomes true when the agent pursing the goal notifies the system of its achievement. For parent goals, achieved is True when the achievement condition of that goal is met. The *obviated* predicate states whether a goal is no longer needed by the system. A goal can be obviate if it is a child of a disjunctive goal that has been achieved. The *preceded* predicate is True if the goal must wait to be assigned due to precedence relations. The *failed* predicate is True if the system has deemed that the goal can never be achieved by the system. These predicates with dynamically runtime set values are investigated in Sections 3.5.5, 3.5.6, and 3.5.7. The goal class predicate and function signatures are listed below.

$$\text{preceded}: G_I \rightarrow \text{Boolean} \qquad \textbf{D66}$$

$$\text{obviated}: G_I \rightarrow \text{Boolean} \qquad \textbf{D67}$$

$$\text{achieved}: G_I \rightarrow \text{Boolean} \qquad \textbf{D68}$$

$$\text{failed}: G_I \rightarrow \text{Boolean} \qquad \textbf{D69}$$

The subgoal^I relation is defined by the system as it creates goal instances. The functions are analogous to their counterparts in G_S they need no additional description. The function signatures are listed below.

$$\text{parent}^I : G_I \rightarrow G_I \qquad \textbf{D70}$$

$$\text{children}^I : G_I \rightarrow P(G_I) \qquad \textbf{D71}$$

$$\text{parent}^{I+} : G_I \rightarrow P(G_I) \qquad \textbf{D72}$$

$$\text{children}^{I+} : G_I \rightarrow P(G_I) \qquad \textbf{D73}$$

$$\text{triggers}^I : G_I \rightarrow P(G_I) \qquad \textbf{D74}$$

$$\text{precedes}^I : G_I \rightarrow P(G_I) \qquad \textbf{D75}$$

The function definitions are listed below. The only assumptions that must be made about these functions, are that (1) when a goal instance is created, the instantiation algorithm correctly specifies the subgoal^I relation and (2) the instanceOf function is mapped properly between G_S and G_I. The other instance tree functions are based on the children^I function, which is based on the subgoal relation. The instanceOf function is used to trace instance goals back to their specification goal.

$\text{parent}^I(g)$	$=\{g_1 : G_I \mid \{g_2,g_3\} \in G_S \wedge \text{instanceOf}(g_1,g_2) \wedge \text{instanceOf}(g,g_3) \wedge \text{parent}(g_2,g_3)\}$	**D76**
$\text{children}^I(g)$	$=\{g_1 : G_I \mid \{g_2,g_3\} \in G_S \wedge \text{instanceOf}(g_1,g_2) \wedge \text{instanceOf}(g,g_3) \wedge \text{subgoal}(g_2,g_3)\}$	**D77**
$\text{triggers}^I(g)$	$=\{g_1 : G_I \mid \{g_2,g_3\} \in G_S \wedge \text{instanceOf}(g_1,g_2) \wedge \text{instanceOf}(g,g_3) \wedge \text{triggers}(g_2,g_3)\}$	**D78**
$\text{parent}^{I+}(g)$	$= \{g_1:G_I \mid (g_1,g) \in \text{subgoal}^I \vee [(g_1,g_2) \in \text{subgoal}^I \wedge g_2 \in \text{parent}^{I+}(g)]\}$	**D79**
$\text{children}^{I+}(g)$	$= \{g_1:G_I \mid (g,g_1) \in \text{subgoal}^I \vee [(g_2,g_1) \in \text{subgoal}^I \wedge g_2 \in \text{children}^{I+}(g)]\}$	**D80**

3.5.3 Additional Instance Tree Relations

There are several additional relations that not defined in the specification tree. These functions are needed in Chapter 4. We will define these relations in this section.

TriggeredParents	$: G_I \rightarrow P(G_I)$	**D81**
LCTP	$: G_I \times G_I \rightarrow G_I$	**D82**
SameSubTree	$: G_I \times G_I \rightarrow \text{Boolean}$	**D83**
Triggerable	$: G_I \rightarrow \text{Boolean}$	**D84**

The function definition for the TriggeredParents gathers the set of goals in the specification tree that are triggered by a sequence of events. The functions sameSubTree and triggerable allow the reasoning to reason over the interactions that goal triggering has on goal precedence. The sameSubTree function takes two instance goal and the function returns True if those goals are in the same instance subtree. The triggerable function returns true if some sequence of events could cause an instance of this goal to be created. The LCTP (Least Common Triggered Parent) function finds the common triggered parent goal that is the lowest in the tree(least). The definition for the TriggeredParents and LCTP are defined below. The definitions for SameSubTree and Triggerable are defined in Section 3.5.6.

TriggeredParents(g)	$=\{g_1 : G_I \mid \{g_2,g_3\} \in G_S \wedge \text{instanceOf}(g,g3) \wedge \text{instanceOf}(g_1,g_2) \wedge g_1 \in \text{parent}^{I+}(g)$ $\wedge\ \text{triggeredBy}(g_3) \neq \{\}\}$	**D85**
$\text{LCTP}(g_1,g_2)$	$=\{g_3 : \text{TriggeredParents}(g_1) \cap \text{TriggeredParents}(g_2) \mid$ $g_4 \in [\text{TriggeredParents}(g_1) \cap \text{TriggeredParents}(g_2)] - g_3 \wedge g_4 \in \text{parent}^I(g_3)\}$	**D86**

3.5.4 Goal Instance Tree Examples

In this section we will show some examples of how the Instance Tree is created through events occurring. We will be using Figure 11 for our specification tree. The Events from E_S use the notation E_X (Uppercase) and the events that occur in E_I use the notation e_X (Lowercase). Also the Instance Tree shown only shows the goals that are not achieved.

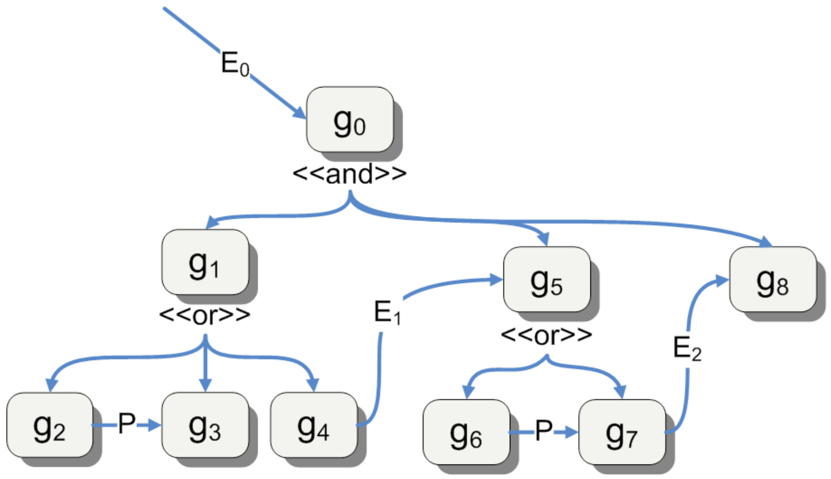


Figure 11 (repeated)

When the system starts the initial trigger occurs. This is the first event of the system. The goals are initially triggered are the ones that do not have explicit triggers and all of their parents do not have any explicit triggers. When the initial triggers occurs the instance is as shown in Figure 13. The diagrams will use the similar style as the specification tree is drawn. The event in E_S that instantiated the instance goal is placed in parenthesis.

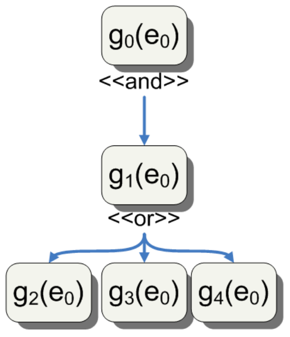


Figure 13. Goal Instance Tree After Initial Triggers

Now suppose that next event is the event specified in Figure 11 is E_1. When a goal in G_S is triggered, instances of its descendents in the form of a subtree are created. This instance subtree recursively adds children to the subtree that are non-triggered subgoals. This event will cause a new instance of g_5 and its subtree to be created. The result of E_1 occurring is shown in Figure 14.

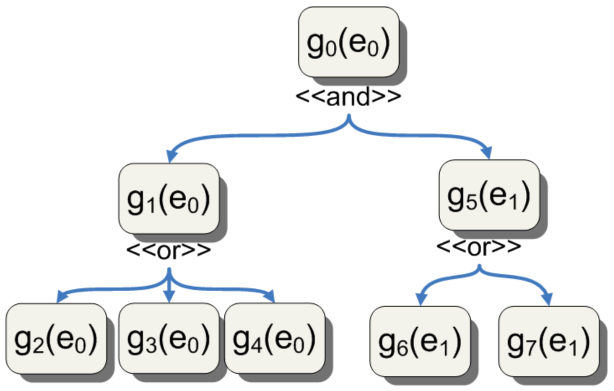


Figure 14. Goal Instance Tree after event E_1

Each one of these instance goals is parameterized with the same parameters its parent goal. In Figure 14, a new subtree containing $g_6(e_1)$ and $g_7(e_1)$ is instantiated when $g_5(e_1)$ is instantiated by event e_1. However if the children have a trigger in the specification tree, there is no need to add those goals to the instance tree. For example, in Figure 13 goal g_8 was not instantiated when goal g_0 was instantiated. This is because goal g_8 has an explicit trigger, and therefore it should not be instantiated by the event that created its' parent. Goal classes that are triggered by another goal are instantiated only when explicitly triggered.

The event E_1 can occur as many times as long as goal g_4 is being worked on by an agent. Our next example (Figure 15) shows another E_1 occurring.

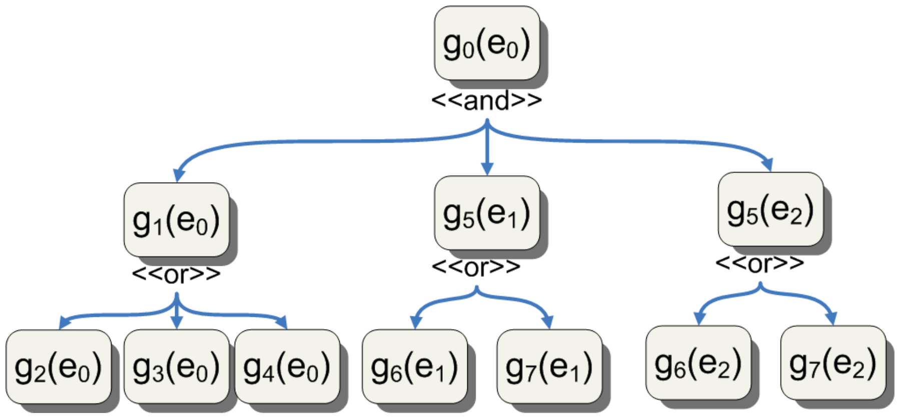


Figure 15. Goal Instnace Tree after 2nd E_1 Event

Now suppose that goal $g_6(e_1)$ is achieved, the goal tree will remove this goal from the instance tree, as shown below (as achieved goals are not shown).

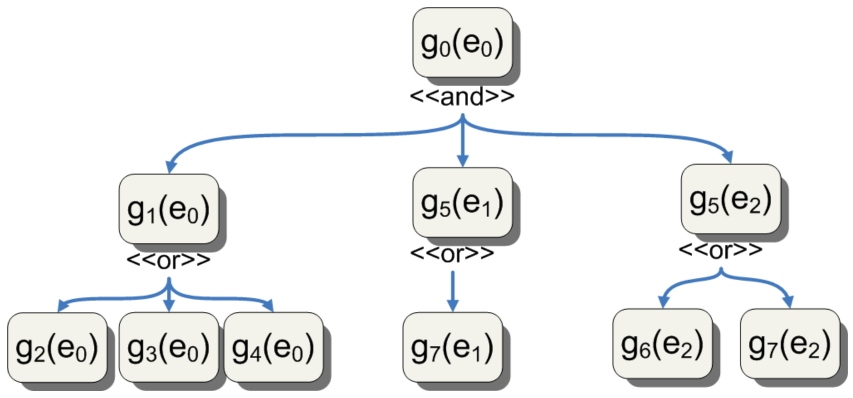


Figure 16. $g_6(e_1)$ Achieved

Another event that is specified in Figure 11 is E_2. This event causes goal g_8 to be created. The result of this event is shown in Figure 17.

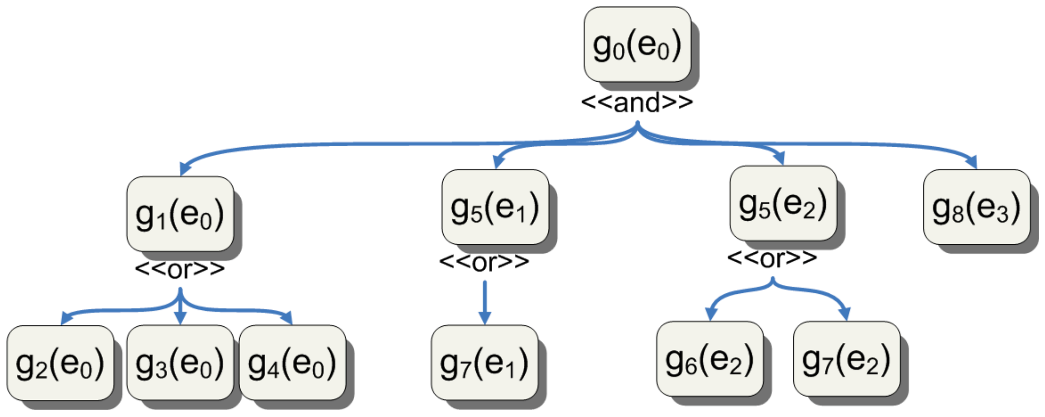


Figure 17. Event E_2 Occured

In Figure 17 the goal subtree creation ensures the formation of a hierarchy of triggered parents. The TriggeredParents function gathers all the parent goals that were instantiated by explicit events from E_I. For example the triggered parents of goal instance $g_7(e_1)$ are triggeredParents($g_7(e_1)$) = $\{g_5(e_1), g_0(e_0)\}$. This function determines the path of instantiation that lead to the instantiation of the goal in question. These local instances are key in extracting other properties of the goal tree, of which are defined later on in the thesis.

3.5.5 Goal Removal

The negative trigger relation allows a goal to be removed from the set of active goals in the instance tree. When a negative triggers occurs, the goal and all its children are removed from G_I. These goals now appear to the system, as if they never existed, and thus the precedence relations will not apply. When a negative trigger removes a goal, all of its children are recursively removed as defined in the following axiom.

$\forall g_1,g_2: G_I \mid \text{removed}(g_1) \wedge g_2 \in \text{children}^{I+}(g_2) \rightarrow \text{removed}(g_2)$ **D87**

3.5.6 Goal Precedence

Goal precedence leads to additional restrictions in G_I. If a goal is preceded by a set of goals, then all of those instances must be achieved before that goal can be worked on. The instance goals that are relevant in the precedence restriction are those that have been instantiated in the same instance sub-tree as the domain goal. Ensuring that LCTP is the same for the instances in the same instance tree and the goal in question is the key to the precedence relation. For example in Figure 15 we will look at goal $g_7(e_1)$. The LCTP between $g_7(e_1)$ and $g_6(e_1)$ is $g_5(e_1)$. The LCTP of $g_7(e_1)$ and $g_6(e_2)$ is $g_0(e_0)$ The precedence for $g_7(e_1)$ is only applicable for $g_6(e_1)$ and not $g_6(e_2)$. Therefore, a function can be defined to check and make sure that two instance goals are in the same instance tree. The sameSubTree function returns True if the first two instance goals are in the same instance tree and if those instance goals are instances of the specification goals in question.

$$\text{sameSubTree}(g_1,g_2) = \{g_3,g_4\} \in G_S \wedge \text{instanceOf}(g_1,g_3) \wedge \text{instanceOf}(g_2,g_4) \wedge \text{instanceOf}(\text{LCTP}(g_1,g_2),\text{LCTP}(g_3,g_4)) \qquad \textbf{D88}$$

In order to check the precedence of a goal, the definition of the triggerable function must be defined. This function takes an instance goal, and gathers all the specification goals that the goal could possibly trigger. This function is defined as follows.

$$\text{triggerable}(g) = \{g_1: G_I \mid \{g_2,g_3\} \in G_S \wedge \text{instanceOf}(g,g_2) \wedge \text{instanceOf}(g_1,g_3) \wedge \text{triggers}^{+}(g_2,g_3) \wedge \text{sameSubTree}(g, g_1)\} \qquad \textbf{D89}$$

The triggerable function makes sure that the two specification goals are elements of the triggers closure, and that the instance goals are in the same subtree. This function definition is needed as some goals could be triggered, and those triggered goals could precede the goal in question. These triggered goals may not exist in G_I, and therefore the function is necessary to locate all of the specification goals that can be triggered from the specific instance goal. To show an example of this function we will slightly modify the the example above. In our modified example, we will add precedence from g_5 to g_8. In the example specification tree in Figure 18 this means that g_8 cannot be worked on until all instances of g_5 are achieved, and no more instances of g_5 can be created.

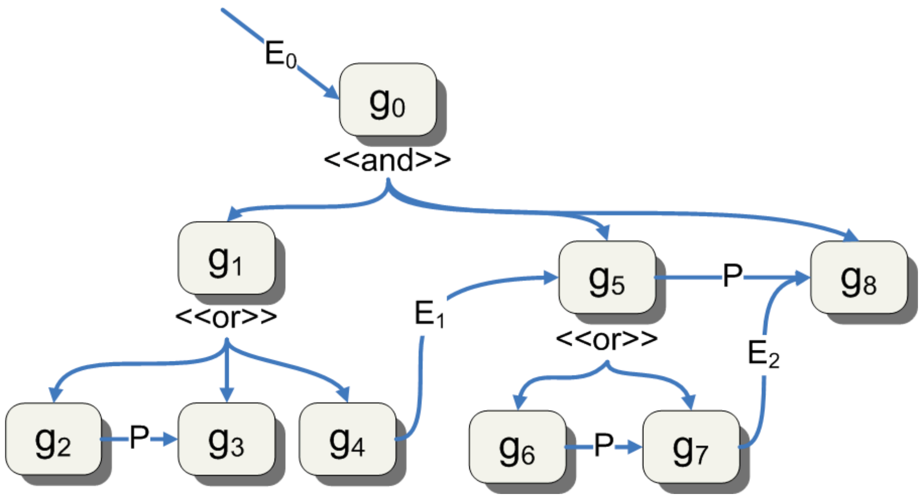


Figure 18. Additional Precedence

When the triggerable function is applied to the goal g_5 in Figure 17 the result will be true. This is because goal g_4 is not achieved and g_4 can trigger a new instance of g_5. When all of the instances of g_4 and g_5 are achieved, then g_8 will not be preceded.

The precedes^I function gathers all the goals in the specification tree that have instances in the instance tree that precede the goal in question or that could create instances that would precede the goal in question. The function chooses the goals only if they are in the same subtree or if the goal is in the triggerable set.

$$\text{precedes}^I(g) = \{g_1 : G_I \mid \{g_2, g_3\} \in G_S \wedge \text{precedes}^+(g_2, g_3) \wedge [\text{sameSubTree}(g, g_1) \vee g_3 \in \text{triggerable}(g_1)]\} \qquad \textbf{D90}$$

With the definition of the triggerable and precedes functions, the preceded predicate can be defined. The preceded function simply checks to see if any of the goals in the precedes set and not achieved or removed.

$$\text{preceded}(g) = \exists\, g_1 : \text{precedes}^I(g) \mid \neg\, \text{achieved}(g_1) \wedge \neg \text{removed}(g_1) \qquad \textbf{D91}$$

3.5.7 Goal Obviation

Disjunctive goals allow choices in the pursuit of goals. The children of an achieved disjunctive goal are no longer needed by the system, and therefore can be labeled as obviated. Therefore when a parent goal is achieved, we define that all of the children that have not been achieved or removed are now obviated. In addition, we want to make sure that we do not add goals to the obviated set if they precede other goals. The definition is as follows.

$$\begin{aligned}\text{obviated}(g) = \; & \exists\, g_1 : G_I \wedge g_1 \in \text{parent}^+(g) \wedge \text{achieved}(g_1) \wedge \neg\text{achieved}(g) \wedge \neg\text{removed}(g) \\ & \wedge \forall\, g_3 \in \text{preceded}^+(g) \mid \neg(\text{active}(g_3) \vee \text{triggered}(g_3))\end{aligned} \qquad \textbf{D92}$$

3.5.8 Goal Achievement

When a goal is checked for achievement there are three condtions to check. If the goal is a leaf goal then that goal can be achieved, and thus we check the achieved function. If the goal is not a leaf goal, then we check the achievement condtion of all of that goals children. If the goal is a disjunctive goal, we check to see if one of the children is achieved. If the goal is a conjunctive goal then we check to see if all of the goals have been achieved or removed from the system.

$$\begin{aligned}\forall g_1 \in G_{I\text{-}Achieved} \Leftrightarrow \quad & (\mathrm{leaf}(g_1) \wedge \mathrm{achieved}(g_1)) \\ & \vee (\exists g_2 \in \mathrm{children}(g_1) \mid \mathrm{disjunctive}(g_1) \wedge \mathrm{achieved}(g_2)) \\ & \vee (\exists g_2 \in \mathrm{children}(g_1) \mid \mathrm{conjunctive}(g_1) \wedge \mathrm{achieved}(g_2) \\ & \qquad \wedge \forall g_3 \in \mathrm{children}(g_1) - g_2 \mid \mathrm{achieved}(g_3) \vee \mathrm{removed}(g_3)) \qquad \textbf{D93}\end{aligned}$$

3.6 Summary

In this chapter, we defined the GMoDS, goal triggering, and goal precedence. We also defined the Goal Specification Tree and the Goal Instance Tree. The definition of the Goal Instance Tree allowed us to define the functions that are necessary for the Execution Model.

Chapter 4 Execution Model

The Execution Model implements GMoDS as defined in Chapter 3. The definition of the model allows it to be analyzed for correctness. The designers should understand the semantics of this model so they can design applications appropriately. The definition of the Execution Model makes its implementation of that model straightforward.

The semantics of the GMoDS Execution Model is based on set theory. In the Execution Model G_I is partitioned into six sets of goals: $G_{I\text{-Triggered}}$, $G_{I\text{-Active}}$, $G_{I\text{-Achieved}}$, $G_{I\text{-Removed}}$, $G_{I\text{-Failed}}$ and $G_{I\text{-Obviated}}$ as shown in Figure 19. Membership in these sets is based upon the set of axioms defined in this chapter. The membership axioms are defined so that the sets partition G_I.

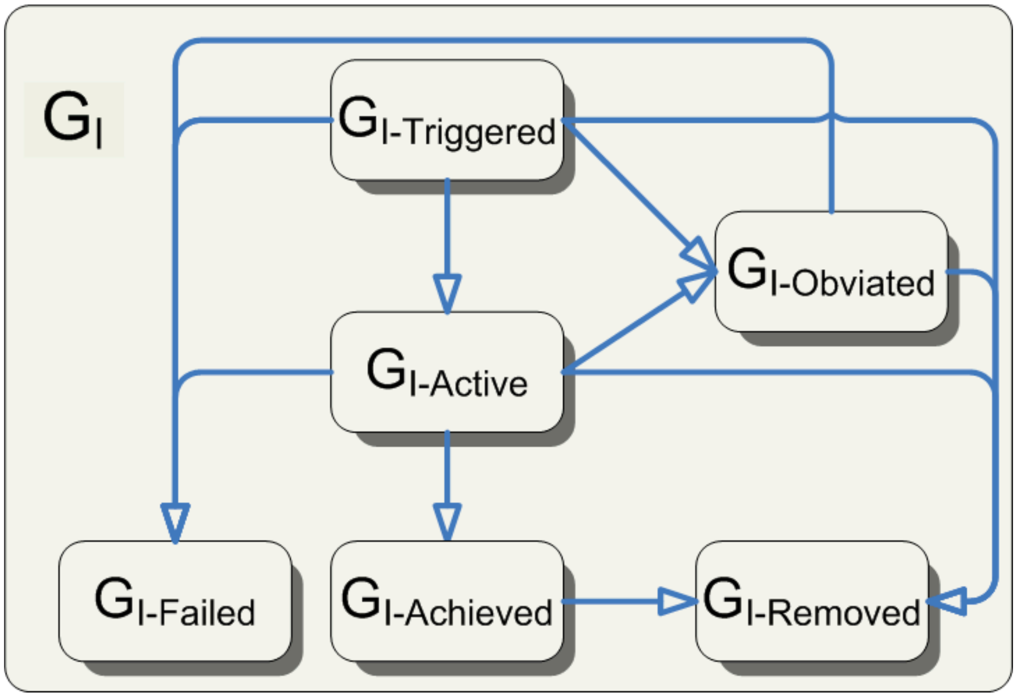


Figure 19. Execution Model

In order to define the movement of goals from one set to another, the functions defined in Chapter 3 are employed. The functions used are preceded, obviated, removed, failed and achieved.

4.1 The Triggered Set

The first set is the triggered set $G_{I\text{-Triggered}}$. This set is where each goal goes once an event that triggers that goal has occurred. Goals stay in this set until one of the other predicates becomes true.

$$\forall\ g : G_I \mid g \in G_{I\text{-Triggered}} \Leftrightarrow \neg\ \text{obviated}(g) \wedge \neg\text{achieved}(g) \wedge \text{preceded}(g) \wedge \neg\text{removed}(g) \wedge \neg\text{failed}(g) \qquad \textbf{D94}$$

4.2 The Active Set

$G_{I\text{-Active}}$ is the set of goals that have been triggered, and the precedence restrictions have been satisfied. All goals that are not preceded by any other goals are allowed to move from $G_{I\text{-Triggered}}$ into $G_{I\text{-Active}}$ Goals in this set remain there until they are either achieved, failed, or obviated. This is defined formally as follows.

$$\forall\ g : G_I \mid g \in G_{I\text{-Active}} \Leftrightarrow \neg\ \text{obviated}(g) \wedge \neg\text{achieved}(g) \wedge \neg\text{preceded}(g) \wedge \neg\text{failed}(g) \qquad \textbf{D95}$$

4.3 The Achieved Set

Ideally, agents should eventually achieve goals that they are working on. When goal achievement does occur, the goal will move from the $G_{I\text{-Active}}$ into $G_{I\text{-Achieved}}$. Agents must notify the GMoDS reasoning when they have achieved their goals, just as they do when a triggering event occurs. Once goals have moved into the Achieved set, they cannot move to any other set. The formal definition is as follows.

$$\forall\ g : G_I \mid g \in G_{I\text{-Achieved}} \Leftrightarrow \text{achieved}(g) \wedge \neg\text{preceded}(g) \wedge \neg\ \text{removed}(g) \wedge \neg\ \text{failed}(g) \qquad \textbf{D96}$$

4.4 The Failed Set

The failed set ($G_{I\text{-Failed}}$) contains goals that the system can never achieve. Once a goal has been put in $G_{I\text{-Failed}}$, that goal may never leave. It is illegal for a failed goal to be removed or achieved, as achievement implies that the system has finished the goal, and removal is as if the goal never existed. The formal definition for $G_{I\text{-Failed}}$ is as follows.

$$\forall\ g : G_I \mid g \in G_{I\text{-Failed}} \Leftrightarrow \text{failed}(g) \wedge \neg\ \text{removed}(g)\ ) \wedge \neg\text{achieved}(g) \qquad \textbf{D97}$$

4.5 The Removed Set

The removed set ($G_{I\text{-Removed}}$) contains goals that have been removed as the result of a negative trigger A goal in the removed set, just as the failed set, cannot be moved to any other set. The decision to remove a goal is application specific, and therefore it does not have a precise definition. When a goal is removed, it is treated as if it never existed in the system. This means that any precedence/triggers relations related to that goal cease to exist when that goal is removed. The formal definition for $G_{I\text{-Removed}}$ is as follows.

$$\forall\, g : G_I \mid g \in G_{I\text{-Removed}} \Leftrightarrow \text{removed}(g) \;\wedge \neg\, \text{failed}(g) \wedge \neg \text{achieved}(g) \qquad \textbf{D98}$$

4.6 The Obviated Set

The obviated set ($G_{I\text{-Obviated}}$) contains goals that no longer need be pursued by the system. These goals are not achieved, and should not be assigned to any agents. The goals in this set may not be removed or failed as discussed above. The formal definition for $G_{I\text{-Obviated}}$ is as follows.

$$\forall\, g : G_{I\text{-Obviated}} \Leftrightarrow \text{obviated}(g) \wedge \neg\, \text{removed}(g) \;\wedge \neg\, \text{failed}(g) \qquad \textbf{D99}$$

4.7 Partition of G_I

In order to prove that the definitions of the Execution Model are correct, we show that each goal can reside in one and only one set (Table 1). This allows the assignment of goals to the sets to be deterministic. The column named *Set* shows the set the goal belongs to and the *Reason* column shows which definition was used to determine set membership.

Case	Failed	Achieved	Preceded	Removed	Obviated	Set	Reason
1	0	0	0	0	0	Active	D95
2	0	0	0	0	1	Obviated	D99
3	0	0	0	1	0	Removed	D98
4	0	0	0	1	1	Removed	D98
5	0	0	1	0	0	Triggered	D94
6	0	0	1	0	1	Obviated	D99
7	0	0	1	1	0	Removed	D98
8	0	0	1	1	1	Removed	D98
9	0	1	0	0	0	Achieved	D96
10	0	1	0	0	1	Achieved	D96
11	0	1	0	1	0	Removed	D98
12	0	1	0	1	1	Removed	D98
13	0	1	1	0	0	Illegal	D94–99
14	0	1	1	0	1	Illegal	D94–99
15	0	1	1	1	0	Illegal	D94–99
16	0	1	1	1	1	Illegal	D94–99
17	1	0	0	0	0	Failed	D97
18	1	0	0	0	1	Failed	D97
19	1	0	0	1	0	Illegal	D94–99
20	1	0	0	1	1	Illegal	D94–99
21	1	0	1	0	0	Failed	D97
22	1	0	1	0	1	Failed	D97
23	1	0	1	1	0	Illegal	D94–99

24	1	0	1	1	1	Illegal	D94–99
25	1	1	0	0	0	Illegal	D94–99
26	1	1	0	0	1	Illegal	D94–99
27	1	1	0	1	0	Illegal	D94–99
28	1	1	0	1	1	Illegal	D94–99
29	1	1	1	0	0	Illegal	D94–99
30	1	1	1	0	1	Illegal	D94–99
31	1	1	1	1	0	Illegal	D94–99
32	1	1	1	1	1	Illegal	D94–99

Table 1

The set listed as *Illegal* fail the conditions stated in D94–99 because either they violate the precedence relation or the goal have been placed in $G_{I\text{-Removed}}$ or $G_{I\text{-Failed}}$. A violation of the precedence is when a goal that is preceded has been achieved. This is clearly a violation of the definition of precedence. Once a goal is removed or failed, that goal cannot be moved to any other set and no agents can work on that goal. Therefore if no agents can work on that goal then axioms cannot change for that goal, and thus the state is illegal. The conflicting conditions are denoted by the gray shading.

4.8 Conclusion

The Execution Model allows the designer to understand properties of the model that must hold during runtime. By looking at the table in Table 1, the designer can see that the model will never move a goal from the removed or failed set, it will not move a goal from the triggered set until precedence is removed or the goal has failed or been removed, etc. These are a few of the guarantees provided by the Execution Model. By understanding the semantic meaning of the Execution Model, the designer can create systems that use goal triggering and goal precedence properly.

Chapter 5 Implementation

The simulator used to demonstrate and evaluate GMoDS is called CROS, which was written in Java. This simulator is designed to simulate multiagent systems that uses the OMACS[12] model for building adaptive multiagent systems. OMACS is used to model organizations and can make use of GMoDS to give it a dynamic model of system goals. Because the implementation of this project must work in tandem with this simulator, Java was the language of choice. Java classes were created to implement all of the entities defined in Chapter 3 (Goals, Events, and Parameters) as well as to manage G_S and G_I. These classes take care of the details of managing both trees, ensuring constraints, and maintaining the relations between the trees.

There were several major challenges that needed addressing. The first was how to manage the sets of goals. The second was the implementation of the preceded predicate (in G_I). The third major challenge was to implement the Execution Model so it runs as efficiently as possible.

5.1 Class Overview

The implementation of the Goal Model closely matched the definitions from Chapter 3 and Chapter 4. Classes were created to implement the main entities as shown in the class diagram in Figure 20. These include the FormalParameters class that implements the Formal Paramaters used for specifying goals and the classes GoalInstance (Instance) and GoalSpecification (Goal). To model the events of the system we defined EventSpec and Event. Each EventSpec has to hold a reference to the Formal Parameters that are required for the creation of a new runtime Event. The runtime Event has to have a reference to the Parameters that are needed in the Goal. To help model the relationships for precedence and triggers, a class for each relationship was created. The top level classes were the Tree structures used for G_S and G_I. The GoalSpecificationTree was created to hold the specification tree, the precedence relations, and the triggers relations. The GoalInstanceTree is used to hold the runtime model of the instance tree. This maintains a set of links (instanceOf), and then the precedence and triggers relations are derived from specification tree.

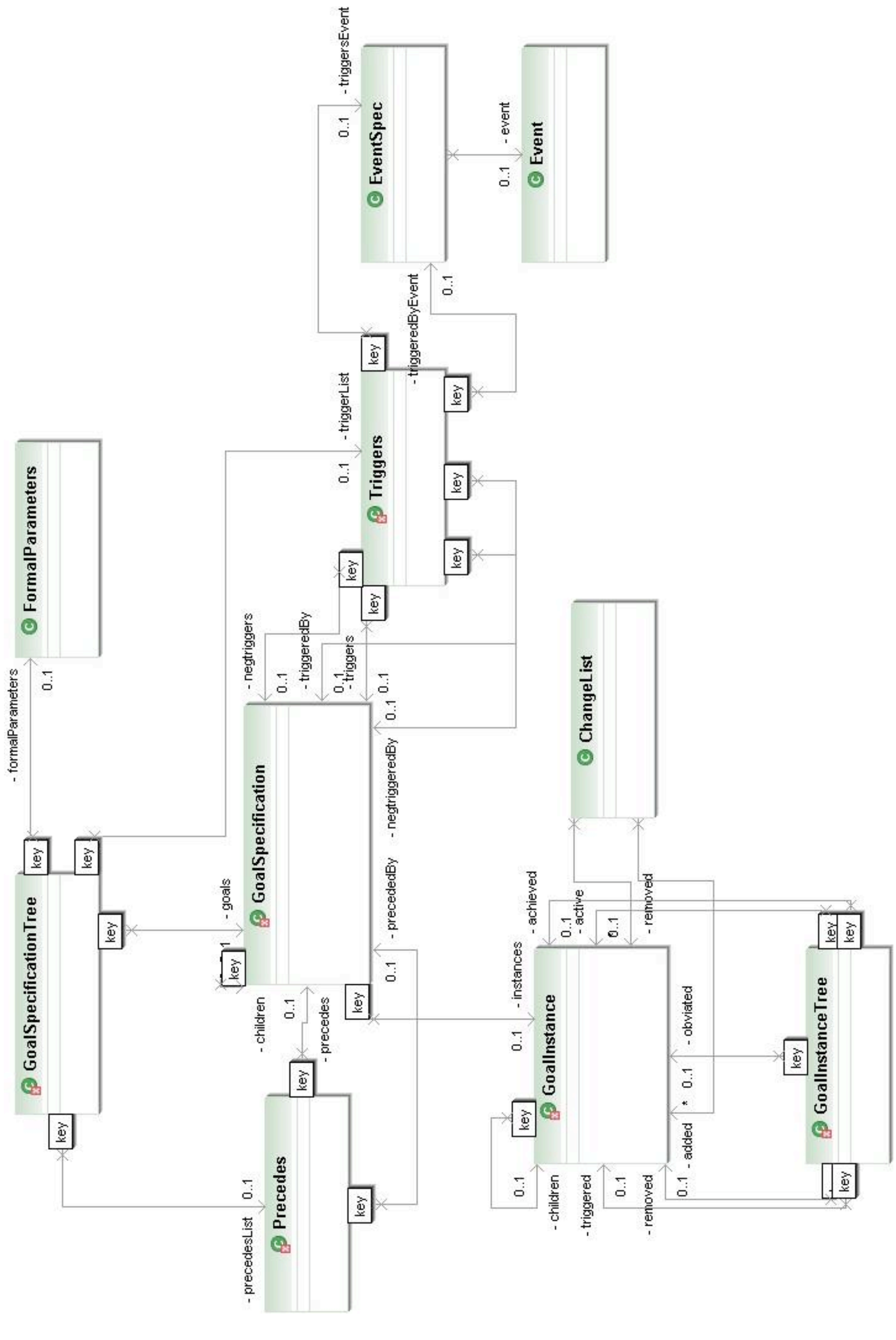


Figure 20. Class Diagram

5.2 Implementation of Goal Instance Tree

The GoalInstanceTree is literally, where the action is. This tree manages the goal instances at runtime. When a Goal Instance Tree is instantiated, a reference to a Goal Specification Tree is required. The Goal Specification tree contains all of the Specification goals, the parent child relationships, the triggers relations, and the precedence relations. In order to keep from reproducing all of this information a single reference to the Specification Tree is used in the Instance Tree. The Specification goals also contain a collection of all the instances that currently exist. The collection of instances that exist, allows the preceded relation to work more efficiently. This collection is updated by the Instance Tree, as this is the only place that can update that information.

There are five sets that to provide instance goal management. These sets are defined in the Execution Model in Chapter 4.

```
private Map<Long, Instance> triggered;
private Map<Long, Instance> active;
private Map<Long, Instance> removed;
private Map<Long, Instance> achieved;
private Map<Long, Instance> obviated;
```

The creation of an Instance starts the process. That instance and all of its children are placed in the triggered set. Then as defined in the semantic model, instance goals and moved from one set to another. The goals are never removed from the tree (via ¬triggers), they are just placed in the removed set. Keeping the goals in the removed set allows the user to perform system traces. An optimized version of this could destroy these goals when they have been removed from the system.

5.3 Goal Precedence

The code for the preceded function code follows the definition in Section 3.5.6. This algorithm is the most complicated part of the implementation. This function uses the functions defined in Chapter 4 to check if another instance could trigger another goal preceding the goal in question.

```
public boolean preceded(GoalInstance goal)
{
SpecificationGoal spec = goal.getSpecificationGoal();
Map<String, SpecificationGoal> specprecedes = goalSpec.getPrecededByPlus(spec);
boolean ret = false;
if (specprecedes.size() > 0)
  {
     Map<Long, Instance> tempIMap = new HashMap<Long, Instance>();
     Map<Long, Instance> checkSet = new HashMap<Long, Instance>();
     for (SpecificationGoal i : specprecedes.values())
          {
```

```
   tempIMap.putAll(i.getInstances());// get all the instances
   for (SpecificationGoal j : goalSpec.getTriggeredByPlus(i, spec).values())
      tempIMap.putAll(j.getInstances());
          }
      for (Instance i : tempIMap.values())//keep the goals in the same subtree
   if (sameSubTree(goal, i))
        checkSet.put(i.getUid(), i);
       for (Instance i : checkSet.values())//gather goals are removed or achieved
         if (achieved.get(i.getUid()) == null && removed.get(i.getUid()) == null)

                          ret = true;
   }
return ret;
}
```

The function first gets all of the specification goals that can precede the goal in question. If that set is non-empty, then all of the instances that are members of the TriggeredBy$^+$ set are added to a set. Then each goal is checked to make sure that that goal is in the same Subtree as the goal in question. The Subtree check ensures that goals in different Subtrees are not erroneously checked. Finally, the remaining goals are checked to ensure that all of those goal are either achieved or removed.

5.4 Dynamic Runtime Environment

The final major and essential challenge was to create a dynamic runtime environment. The implementation in the Instance Tree was clearly the best place to implement this.

5.4.1 Event Occurrence

Once the instance tree is created the only operations that are permitted are event based. These events are of the type EventSpec, and they have been defined in the GoalSpecificationTree. There are two functions that are publicly available via the AbstactGoalInstanceTree interface shown below.

```
public interface AbstractGoalInstanceTree
{
   public abstract<T> ChangeList occured(Event e);
   public abstract<T> ChangeList initalTriggers(T params);
}
```

There are the initial triggers and the occurred method. The initial trigger method allows the goal tree to be instantiated. The occurred method allows an event to occur. When an event has occurred, the goal tree is updated accordingly. The goal tree checks to see what type of event has occurred, and then it acts appropriately. When a goal is triggered (isPositive), the goal tree creates the proper new subtree and all of the new goals will start out being placed it the Triggered set. When a negative trigger occurs (isNegative) the goals that need to be removed are placed in the Removed set via the remove method. When a goal is achieved (isAchieved) that goal is move to the achieved set, and then the achievement of that goals parents are recursively checked based on the achievement conditions that have been specified in the goal specification tree.

```
public<T> ChangeList occured(Event e)
  {
    diff.beforeState(this.getLeafGoals());
    GoalInstance instance = e.getInstance();
    EventSpec specevent = e.getEvent();
    if(instance != null)
    {
      if(specevent.isAchieved())
      {
     achieved(e.getInstance());
      }
      else if (specevent.isPositive())//an event with a goal
      {
    boolean done = false;
    GoalInstance Parent = null;
         Map<String, GoalSpecification> triggers;
    triggers = new HashMap<String, GoalSpecification>();
    GoalSpecification specGoal= instance.getSpecificationGoal();
    while(! done)
       {
      if (instance.getInstanceParent() != null)
         Parent = triggeredParent(instance.getInstanceParent());
      else
         Parent = triggeredParent(instance);
      triggers = this.goalSpec.getTriggers(specGoal, specevent);
      if(!triggers.isEmpty() || specGoal.getSpecParent() == null)
         done = true;
      else
         {//code for moving up the tree, if a triggers occurs in a parent
         instance = Parent;
         specGoal = specGoal.getSpecParent();
         }
       }
         GoalInstance specInstanceParent;
     for (GoalSpecification t : triggers.values())
       {
           specInstanceParent =
                             getInstanceFromTriggeredParent(Parent, t.getSpecParent());
      GoalInstance newOne = createTree(t, specInstanceParent, OTHER, e);
      specInstanceParent.addChild(newOne);
      triggered.put(newOne.getUid(), newOne);
       }
      }
      else if (specevent.isNegitive())
      {
     GoalSpecification instanceSpec = instance.getSpecificationGoal();
     Object type = e.getParams();
     Map<Long, GoalInstance> list = this.getInstanciated();
     for (GoalInstance l : list.values())
       {
        if( l != null)
      if (l.getParameter().equals(type) &&
                  goalSpec.getTriggers(instanceSpec,specevent).containsKey(l.getName()))
         remove(l);
           }
      }
      else
      {
     System.err.println("Error Event not found");
      }
    }
    else
    {
   System.err.println("instance is null");
```

```
     }
 checkSets();
 diff.afterState(this.getLeafGoals());
 printSnapShot();
 return diff;
 }
```

5.4.1.1 The ChangeList

In order to make working with the goal model easier, the Instance Tree returns a ChangeList when the initial triggers method or the occurred method is called. The ChangeList contains two sets, the removed and the added. The removed set is the set of active leaf goals that have been removed from the $G_{I\text{-}Achieved}$ due to the event occurring. The added set contains the leaf goals that have been activated and added to $G_{I\text{-}Achieved}$ due to the event occurring. This may be due to new goals being instantiated, or it may be from a goal being achieved and then the goal precedence has been relieved. The actual algorithm uses set difference to calculate both sets based on the leaf goals that exists in the pre state and the post state. If the two sets are the same, then we just return empty sets for both sets.

```
 protected void computeDifference()
    {
       Collection<GoalInstance> temp;
                temp  = new LinkedList<GoalInstance>(added);
       if(! removed.equals(added))
       {
          added.removeAll(removed);//new
          removed.removeAll(temp);//removed
       }
       else
       {
          added = new LinkedList<GoalInstance>();
          removed = new LinkedList<GoalInstance>();
       }
    }
```

5.4.2 Achievement Actions

When goals are achieved then the Execution Model needs to update the instance tree to take reflect the changes that have occurred in the system. These updates include moving goals into the achieved set and moving goals from the triggered set into the active set, and moving obviated goals. The first action is to check for the completion of parent goals. This check is recursive, and continues while goals are being moved into $G_{I\text{-}Achieved}$, or when the root goal is moved into $G_{I\text{-}Achieved}$. The code for doing this check is listed below.

```
private void recursiveAchieved(GoalInstance instance){
  if (instance != null) {
    GoalInstance temp,tempparent;
    Long uid = instance.getUid();
    temp = findAndRemove(uid);
    achieved.put(uid, temp);
    tempparent = instance.getInstanceParent();
    if(tempparent != null)//non-root {
      if(instance.getSpecificationGoal().isDisjunctive) {
        Map<Long,GoalInstance> removeAbleChildren;
        removeAbleChildren = new HashMap<Long, GoalInstance>();
        //get all children
        for(Long key: instance.getChildren().keySet()) {
          temp = findAndRemove(key);
          if(temp != null)
             removeAbleChildren.put(key, temp);
        }
        for(Long key : removeAbleChildren.keySet())
        obviated.put(key, removeAbleChildren.get(key));
        GoalSpecification parent = tempparent.getSpecificationGoal();
        if(parent.isDisjunctive())
          recursiveAchieved(tempparent);//recursive
      }
      else if(parent.isConjunctive()) {
        Map<Long,GoalInstance> children = tempparent.getChildren();
        boolean allAchieved = true;
        for(GoalInstance child : children.values()){
          if(achieved.get(child.getUid()) == null)
            allAchieved = false;
        }
        if(allAchieved)
          recursiveAchieved(tempparent);//recursive call
      }
      else
        System.exit(-1);
    }
    else //root is achieved
      rootIsAchieved = true;
  }
}
```

This recursive check allows the reasoning to ensure that all high level goals are in the achieved set if their achievement condition has been met. While traversing up the tree, the organization will move obviated goals into $G_{I\text{-}Obviated}$. The call for this is done in the isDisjunctive branch of the recursive check when the current goal is disjunctive and it has been achieved. In both branches, a recursive call is done if the achievement condition of the parent has been satisfied. These goals are the children of an achieved disjunctive goal as defined above. When one goal is moved to $G_{I\text{-}Obviated}$ then all of the descendants of the goal are obviated, if they are not removed or achieved. The third action is to check all goals that are in $G_{I\text{-}Triggered}$ for the removal of precedence restrictions.

```
public void checkPrecedence()
   {
   Map<Long, GoalInstance> moveList = new HashMap<Long, GoalInstance>();
   GoalInstance temp;
   for (Long key : triggered.keySet())
      {
         temp = triggered.get(key);
         if (!preceded(temp))
            moveList.put(key, temp);
      }
   for (Long key : moveList.keySet())
      {
         temp = triggered.remove(key);
         active.put(key, temp);
      }
   }
```

5.5 Algorithm Complexity

An algorithm complexity analysis was preformed on the implementation. When an event occurs, the complexity of the algorithm is entirely based on the size of the goal tree (in the worst case). Suppose that the size of the instance goal tree is *n*, the size of the eventSpec is *t*, and the height of the tree is *h*. The complex operations run in O(*t***n*) or O(h*n) time. These operations include recursing up the goal tree to check achievement conditions, the creation of new goals, and checking precedence relations.

```
public<T> ChangeList occured(Event e)
   {
      diff.beforeState(this.getLeafGoals());***************N****************
      ***************CONSTANT****************
      if(instance != null)
      {
         if(specevent.isAchieved())
            achieved(e.getInstance());***************N****************
         else if (specevent.isPositive())
         {
        ***************CONSTANT****************
        if (instance.getInstanceParent() != null)
           ***************CONSTANT****************
                   else
           ***************CONSTANT****************
        for (SpecificationGoal t : triggers.values())******T******
        {
            Instance specInstanceParent = this.getInstanceFromTriggeredParent(***);
            Instance newOne = createTree(t, specInstanceParent, OTHER, e); ****N*****
            ***************CONSTANT****************
        }
        else if (specevent.isNegitive())
           {
           ***************CONSTANT****************
           Map<Long, Instance> list = this.getInstanciated();*********CONSTANT ******
           for (Instance l : list.values())***************N ****************
             {
             if( l != null)
                 ***************CONSTANT****************
             else
                 ***************CONSTANT****************
             }
           }
      else
          ***************CONSTANT****************

      checkSets();***************(NH)****************
      diff.afterState(this.getLeafGoals());***************N****************
      printSnapShot();***************N****************
      return diff;
}
```

The code above has been annotated with the amount of time each operation takes. The code that contains only constant time operations was removed and labeled with CONSTANT. The other code that has run times that are not constant are labeled with their bound. In the positive triggers section the code runs in O(n*t) time. Also the checkSets() function takes O(h*n) time. If both of these operations take place, then the bound on the running time of the occurred method is O(h*n) + O(n*t). This is in the worst case, and trees that are more properly constructed will clearly take much less time to run. As this is an over estimate, the actual time is much less, due to the fact that as average height of node in the tree increases, the number of goals that can be triggered by a single specification goal decreases, and therefore the typical runtime bound would be closer to O(max(h,t)*n). This decrease is due to constraints that have been placed on the manner in which the goal tree is constructed (no precedence cycles, and restrictions on how triggers may be constructed).

5.6 Implementation Conclusions

The implementation is directly derived from the definitions that were needed to properly define a dynamic Execution Model. This Execution Model allows a system to interact with the goal model in a abstract manner. The system only need only create the specification model and then call the initialTriggers method. Then it can then call the occurred method each time that an event occurs and update the list of leaf goals via the ChangeList structure that is returned via both the initialTriggers and the occurred methods.

The runtime system also ensures that the sets defined in Chapter 4 are properly maintained. The implementation also insures that the only way to manipulate the sets is through the public initialTriggers and occurred methods.

Chapter 6 Results

Our approach to demonstrating the working Execution Model was to use the model in an application. After the model was executed, we examined snapshots of the system to verify that the model has sustained the properties of the model that we have specified (specifically triggers and precedence).

The results of the simulation are shown graphically through snapshots. Each time the *occurred* method was called, a snapshot of the system was printed. This snap shot allows the user of the user to observe what is happening to the goal model when the event has occurred. These snapshots are incremental and thus allows the user to see the state of the goal model during execution.

6.1 The Application

The application used to test the Execution Model attempts to find, detect, and remove weapons of mass destruction. There are several robots (agents) that are placed in a room. The robots have to search the room looking for unknown objects. When these unknown objects are found the agents will try to determine if the objects are weapons. The agent can detect three types of weapons; Biological Weapons, Chemical Weapons, and Nuclear Weapons. We have designed the system such that there is one agent that can play each of these roles. Each weapon can be of one type. If the object is tested for one type and the test fails, then the other types need to be tested. If the test does not fail and the object is a weapon, then the other tests are unnecessary and the weapon needs to be moved to a safe location.

6.2 The Dynamic Execution Model

The first snapshot we will look at is the Specification of the Goal model that is in the GoalSpecificationTree. Two versions of the implementation were tested. The first was the actual specification that was elicited in the requirements phase with precedence. This version has many triggers relations and a precedence relation. Therefore, to test the effect of precedence relations, we also created a version with precedence. We discuss both of these examples to illustrate how the model works.

6.2.1 Specification Tree Goal Design

We can see in Figure 21 that there is one top level goal which is to perform a WMDSearch. This is the goal of the system, to find WMD. There are two subgoals below that named FindWMD and RemoveLoc. FindWMD is a goal to find the Weapons, and RemoveLoc is a goal to move located weapons into a safe place. The FindWMD is broken down into four additional sub-goals. The first goal is Initialize, which initializes the system. The second goal is AssignArea, which allows the system to divide up the region in question into several partitioned areas to be searched. The SearchArea goal is to search a region for the weapons. If objects are found then an IdentifyObjects goal is triggered. This goal categorizes the weapons by their type. The IdentifyObjects goal has three subgoals: CheckChem, CheckRadio, and CheckBio. These goals have the specific purpose of identifying a specific type of weapon. These goals are mutually exclusive in that once the weapon has been detected as one type, the goals that remain to be checked are no longer needed.

6.2.1.1 Triggers and Precedence

The triggers relations are denoted in Figure 21 by an arrow with an event and parameters in parenthesis. The goals Initalize, AssignArea, SearchArea and IdentifyObjects are triggered by different events (assign, search, and objectFound). The RemoveLoc goal is triggered by the IdentifyObjects goal when a weapon has been detected. To produce the goal removal of the children of IdentifyObjects a negative triggers is placed from each of the children to the other two children. These negative triggers are identified by a dotted line with the event and parameters in parenthesis. The RemoveLoc is preceded by the IdentifyObjects goal. The precedence relation is denoted in Figure 21 by the line with the label <<**Precedes**>>. The meaning of the precedence is that the system will wait to assign any agents until all of the IdentifyObjects have been achieved.

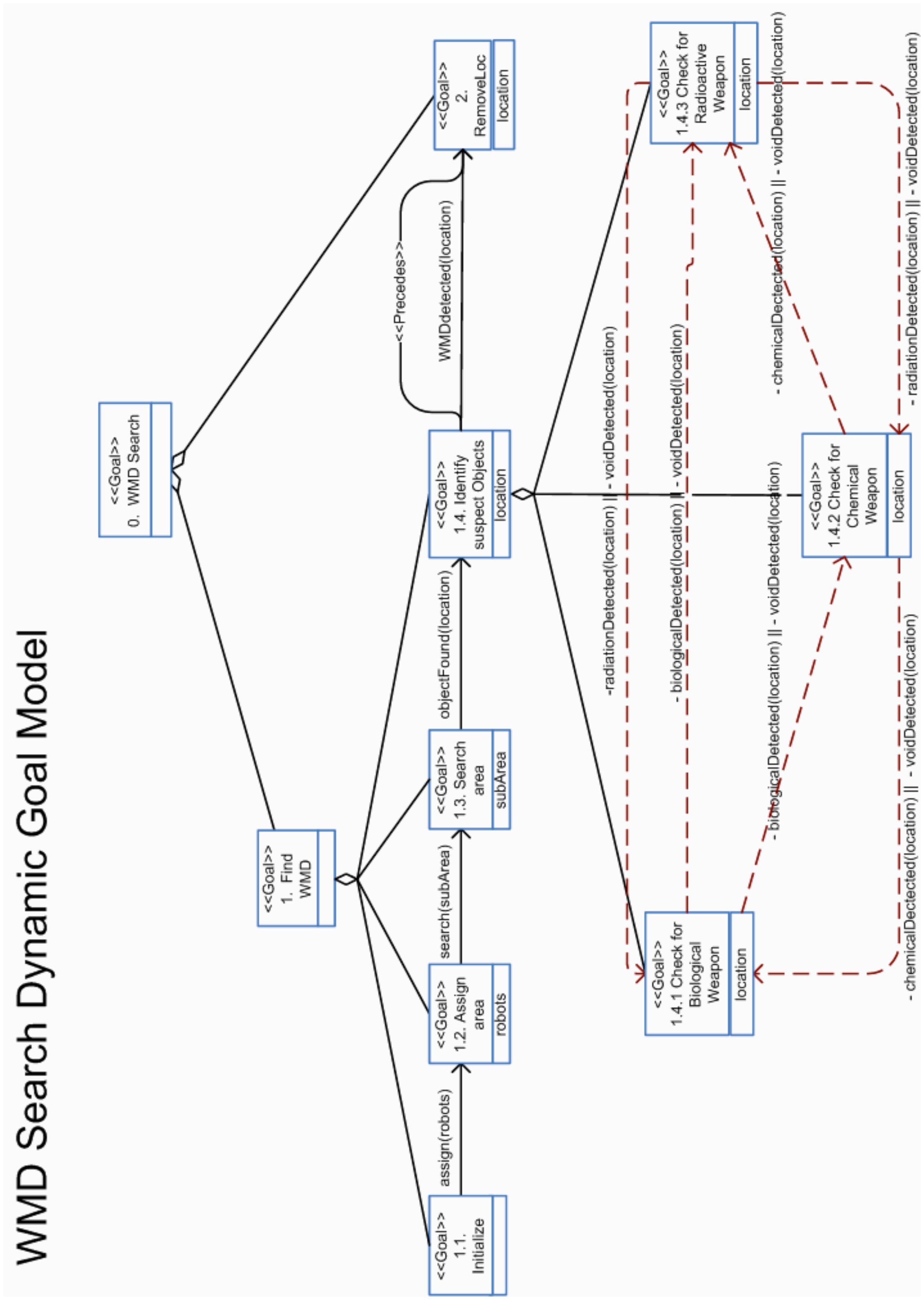

WMD Search Dynamic Goal Model
<<Goal>> 0. WMD Search
<<Goal>> 1. Find WMD
<<Goal>> 2. RemoveLoc
location
<<Goal>> 1.1. Initialize
assign(robots)
<<Goal>> 1.2. Assign area
robots
search(subArea)
<<Goal>> 1.3. Search area
subArea
objectFound(location)
<<Goal>> 1.4. Identify suspect Objects
location
<<Precedes>>
WMDdetected(location)
<<Goal>> 1.4.1 Check for Biological Weapon
location
<<Goal>> 1.4.2 Check for Chemical Weapon
location
<<Goal>> 1.4.3 Check for Radioactive Weapon
location
-radiationDetected(location) || - voidDetected(location)
- biologicalDetected(location) || - voidDetected(location)
- biologicalDetected(location) || - voidDetected(location)
- chemicalDectected(location) || - voidDetected(location)
- chemicalDectected(location) || - voidDetected(location)
- radiationDetected(location) || - voidDetected(location)

Figure 21. Goal Specification Tree

6.2.2 Goal Instance Tree

The Goal Instance Tree implements the Goal Specification Tree and is at runtime. A snapshot of the Goal Instance Tree was produced each time an event happens occurred that changed the goal tree. During the runtime of the system there were 161 snapshots produced. For the sake of brevity, we will only show a portion of snapshots to demonstrate correctness. The shown in the snapshots are the goals in the triggered and active sets only. The other sets are not shown as they increase the size of the figures and do not provide much insight as to how the system operates. In addition we also show the goals as a tree. This allows the user to easily compare the snapshot to the specification tree. All of the snapshots are published on the website associated with this thesis (http://macr.cis.ksu.edu/GMoDS/). In the snapshots below the top oval shows the event that has occurred. The parenthesis show the parameters used to create that goal. After the event name or the parameters is a snapshot number. This snapshot number provides a chronological ordering to the events. The arrows between the goals shows the subgoal relation. In the simulation, all of the triggers events are labeled with assign and not the specific events that occurred. To clarify this issue, the text will clearly state which event caused the snapshot to be taken.

In Figure 22, the initial triggers occurs, and the only goals that are created are the WMDSearch, FindWMD, and Initalize. These are the goals that are not triggered by any other goals. In Figure 23 we can see that an assign event has occurred during the pursuit of the Initialize goal. The system triggers an AssignArea goal, and the effect of this event is reflected in Figure 23. In Figure 24 we can see that an achieved goal has occurred. We can see from the difference in Figure 23 and Figure 24 that the goal that the Initialize goal was achieved and thus removed from the snapshot view. In Figure 25 a search event has occurred. This event triggers a searchArea goal.

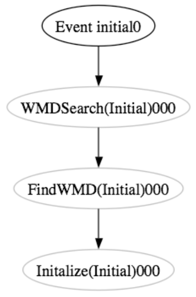


Figure 22. Initial Triggers

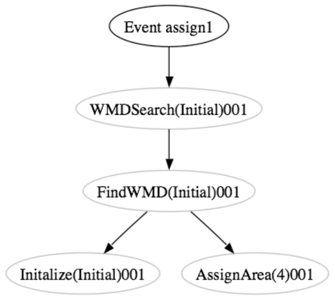


Figure 23. Assign Area Triggered

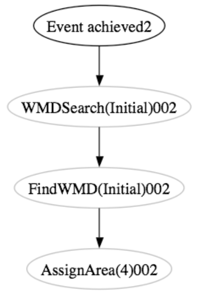


Figure 24. Initialization achieved

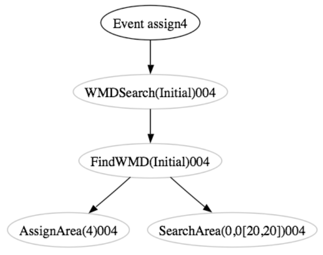


Figure 25. Search Area Triggered

In Figure 26 the AssignArea goal is achieved and removed from the system. In Figure 27 an IdentifyObjects goal has been triggered by the objectFound event. This example shows that the children of the IdentifyObjects goal have also been created as defined in Section 3.5.4 .

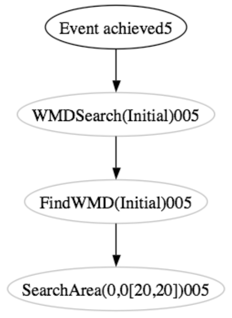


Figure 26. Assign Area Achieved

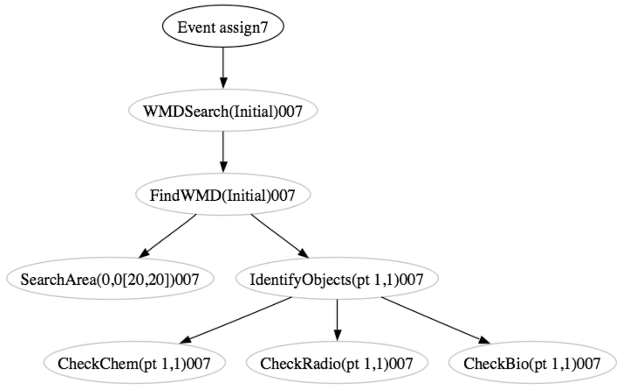


Figure 27. Identify Objects Triggered

In Figure 28 we can see that another IdentifyObjects goal has been triggered by an objectFound event. The additional IdentifyObjects triggering shows that the goal model allows multiple instances of the same specification goal. This shows that the system is dynamic in nature, yet conforms to the specification given in the Specification Tree.

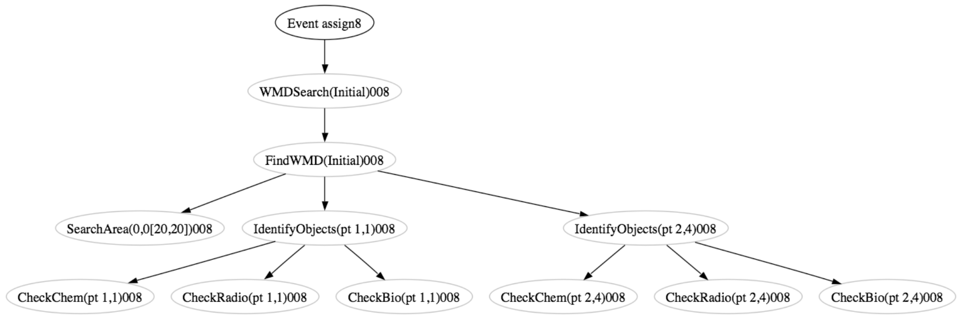


Figure 28. Another Identify Objects Triggered

In Figure 29 a CheckBio(1,1) goal has been achieved. We can see that an achieved event has occurred and that the CheckBio(1,1) goal has been removed from the snapshot of the system.

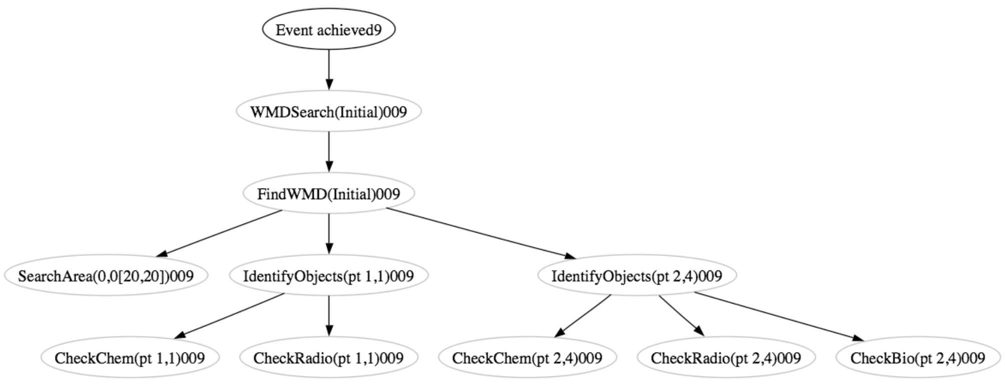


Figure 29. CheckBio(1,1) is achieved

In Figure 30 a RemoveLoc goal has been triggered by a WMDdetected event, which cannot be worked on until all of the IdentifyObjects goals have been achieved (as defined by the precedence). If the diagram were in color, the RemoveLoc would be colored Blue, which means that it was preceded. All of the other goals would be colored green, which indicates that they are active.

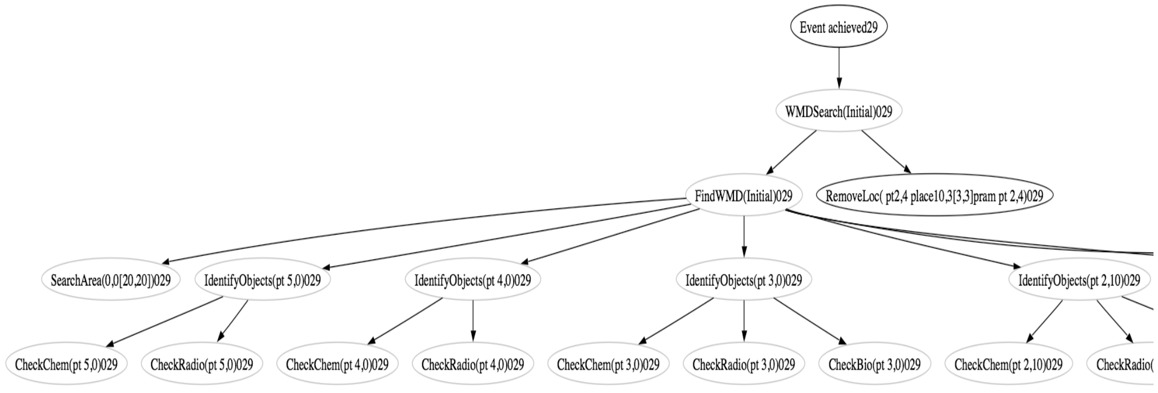


Figure 30. RemoveLoc Goal Created

There are several snapshots between Figure 30 and Figure 31. The first case is that a child of the IdentifyObjects triggered a RemoveLoc, and then the remaining children were negative triggered. The second case was that all of those goals were achieved and thus the IdentifyObjects goal was achieved. In Figure 31 we can see that all of the IdentifyObject goals have been achieved; however, the 3 RemoveLoc goals are still preceded as the SearchArea goal has not been achieved and it could trigger another IdentifyObjects goal.

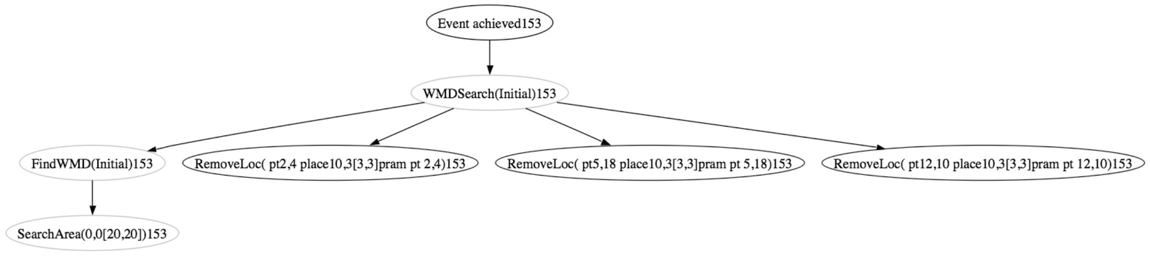

Event achieved153
WMDSearch(Initial)153
FindWMD(Initial)153
RemoveLoc(pt2,4 place10,3[3,3]pram pt 2,4)153
RemoveLoc(pt5,18 place10,3[3,3]pram pt 5,18)153
RemoveLoc(pt12,10 place10,3[3,3]pram pt 12,10)153
SearchArea(0,0[20,20])153

Figure 31. No more IdentifyObject goals

When the SearchArea goal is achieved as shown in Figure 32, then the RemoveLoc goals are no longer preceded. This is denoted in the snapshots by a change in color, which is not present in this document. Now agents are allowed to be assigned to work on these goals.

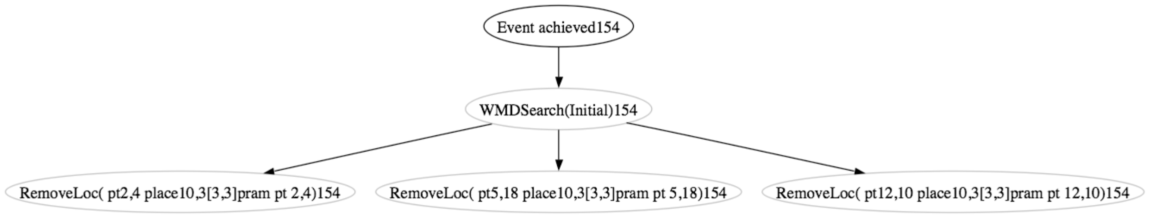

Event achieved154
WMDSearch(Initial)154
RemoveLoc(pt2,4 place10,3[3,3]pram pt 2,4)154
RemoveLoc(pt5,18 place10,3[3,3]pram pt 5,18)154
RemoveLoc(pt12,10 place10,3[3,3]pram pt 12,10)154

Figure 32. SearchArea Achieved

In the Figure 33 it is clear that all of the other goals in the system have been achieved. The system is now ready to terminate.

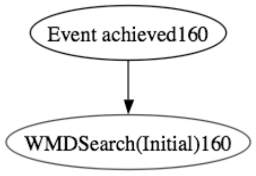


Figure 33. WMDSearch Achieved

6.2.3 Goal Instance Tree Without Precedence

The preceding example showed that the precedence relations were implemented correctly. We now show the same system, without the precedence relations. The only difference between the specification tree in Figure 21 and the one in Figure 34 is the absence of the precedence relation.

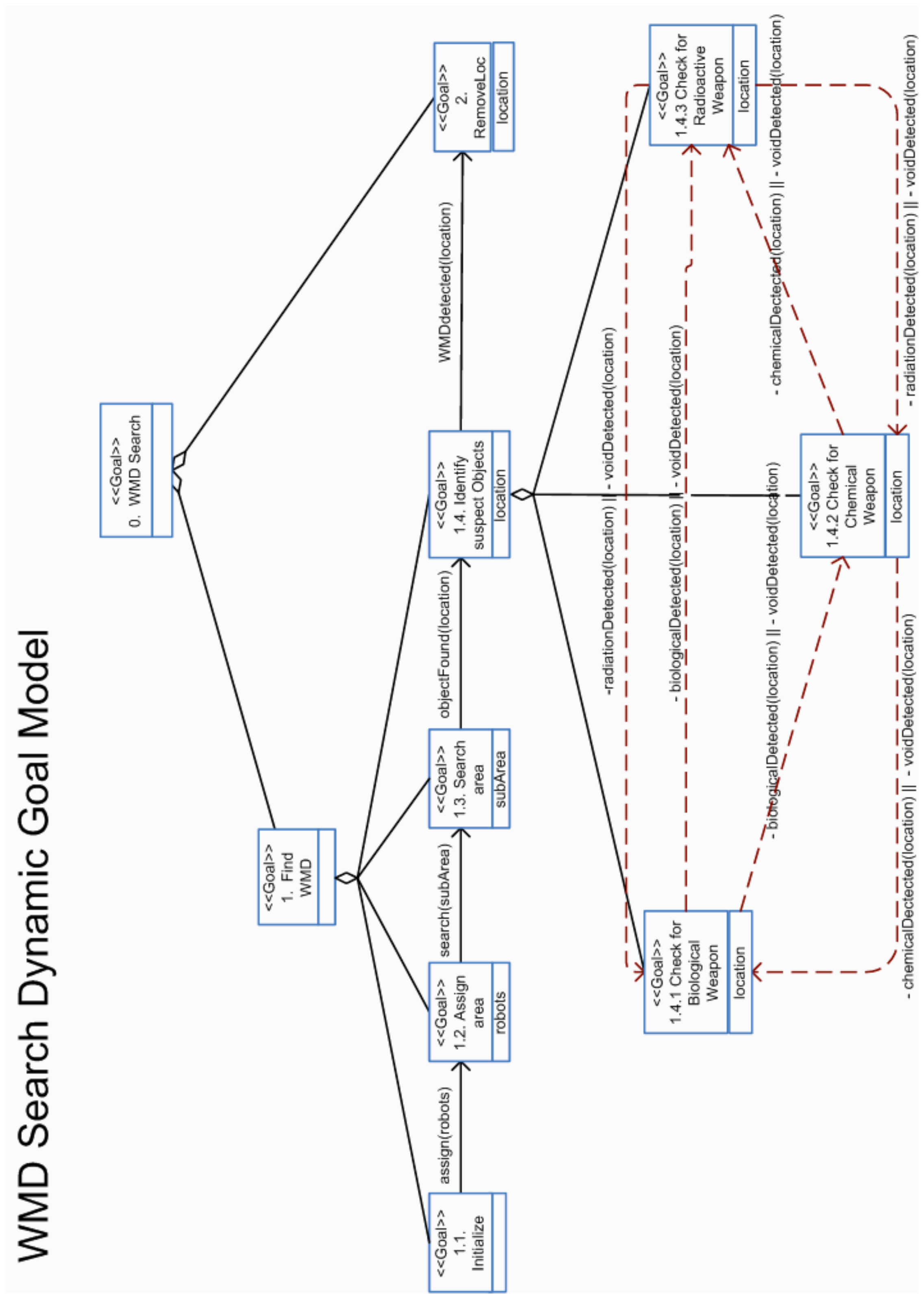

WMD Search Dynamic Goal Model
<<Goal>> 0. WMD Search
<<Goal>> 1. Find WMD
<<Goal>> 2. RemoveLoc
location
<<Goal>> 1.1. Initialize
<<Goal>> 1.2. Assign area
robots
<<Goal>> 1.3. Search area
subArea
<<Goal>> 1.4. Identify suspect Objects
location
<<Goal>> 1.4.1 Check for Biological Weapon
location
<<Goal>> 1.4.2 Check for Chemical Weapon
location
<<Goal>> 1.4.3 Check for Radioactive Weapon
location
assign(robots)
search(subArea)
objectFound(location)
WMDdetected(location)
-radiationDetected(location) || - voidDetected(location)
- biologicalDetected(location) || - voidDetected(location)
- biologicalDetected(location) || - voidDetected(location)
- chemicalDectected(location) || - voidDetected(location)
- radiationDetected(location) || - voidDetected(location)
- chemicalDectected(location) || - voidDetected(location)

Figure 34. G_S without precedence

We start in the middle of the runtime of the system for the new specification tree as the beginning snapshots are identical to the previous example. The snapshot that is equivalent to the one in Figure 35 is Figure 30. In Figure 35, a RemoveLoc goal that has been triggered. This goal is not preceded by any other goals and therefore the system can assign an agent to work on that goal.

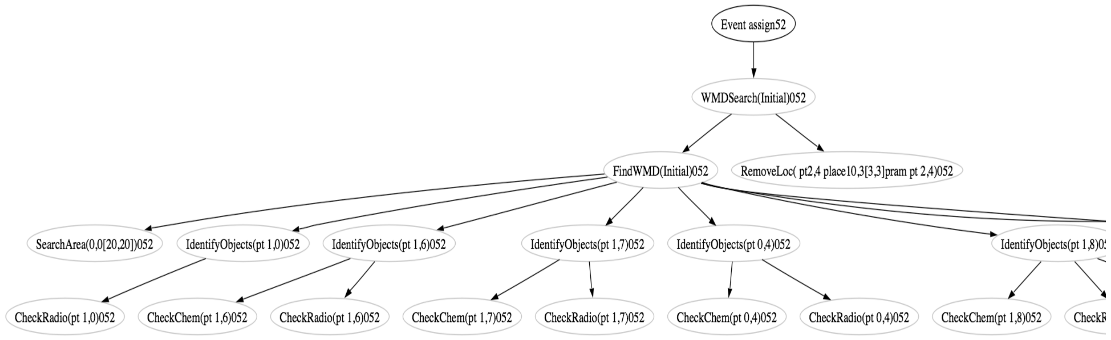


Figure 35. RemovedLoc Goal exists during runtime

In Figure 36, we can see that the RemoveLoc goal has been achieved, which follows the behavior specified in Figure 34.

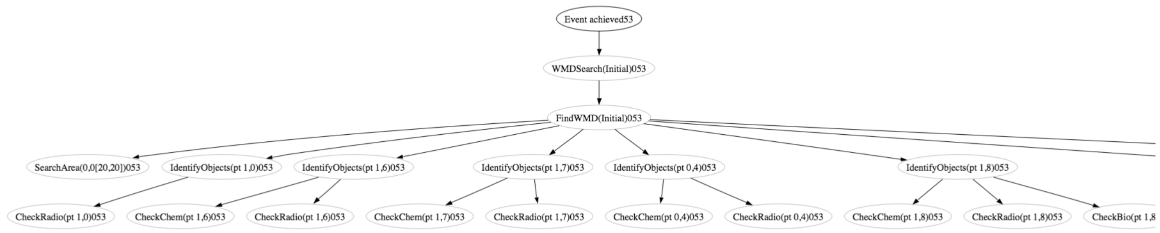


Figure 36. RemoveLoc Achieved

The snapshots that follow Figure 36 behave in the same manner as the sequence of snapshots listed in this section.

6.3 Conclusion

From these examples, it appears that the demonstration system correctly implements the goal model as defined in Chapter 3 and Chapter 4. The specification model allows goal decomposition via and/or relations and the elicitation of goal triggers and goal precedence. The goal model handles the initial trigger, goal achievement, goal triggering, and precedence relations, which are the critical elements that the Execution Model needs to handle.

Chapter 7 Conclusions and Future Work

There are several goal frameworks that have been developed for multi-agent systems and each of these models is good at modeling a static systems. However, none of these models provide end-to-end modeling with invariant assurance. GMoDS, which can dynamically adapt to the environment, provides end-to-end goal modeling, and that can ensure runtime invariants, a significant advancement for the multi-agent community. We formally defined the GMoDS specification model, which included goals, events, goal precedence, and goal triggers. Goal precedence allows an ordering of goal achievement and goal triggers allows a goal to create additional goals or remove current goals in the system. Along with the specification model, GMoDS defines an instance model to represent the current state of the system at runtime. The instance model maintains the structure defined by the specification model, and yet provides the ability to change dynamically. The instance model uses the formal definitions of the specification model to ensure that goal triggering and goal precedence are upheld during the runtime of the system. We also defined an Execution Model that implements the instance model during the runtime of the system. The Execution Model maintains a group of partitioned sets that represent goals that are in similar states. The implementation provides a proof of concept that implements the entities defined in the specification model and the instance model. The implementation maintains the Execution Model sets and thus ensures that the properties defined in the specification model and instance model are upheld.

The GMoDS framework provides a goal modeler with a guarantee on how goals are triggered and on how goal precedence works at system runtime. System designers can leverage this knowledge to design better models. Designers can also see how their model will work by running a simulation, using the Execution Model code. The Execution Model is also being incorporated into an agent/robot architecture for controlling teams of agents/robots.

Future work on GMoDS should address developing additional properties for goals such as softgoals, goal preference, goal policies, and the use of goal metrics. Soft goals are goals that have non-functional requirements. They are items like “make the customer happy”, “increase profits”, or “don’t waste time”. These goals are useful to the system but they typically cannot be directly achieved. Agents can work on hard-goals in order to try to satisfy some softgoals.

Goal preferences allow the designer to give the system a hint into what goals are preferred when the system is given a choice. For example, this is useful when the system has to select a goal from a set of disjunctive children. A preference value could be used to guide the system to choose the goal that the designer prefers.

Goal metrics allow the goal model to be quantifiably be measured. Such measurement would allow the designer to gauge how well the model will work in real system. Metrics could be based on running the goal tree through a mathematical analysis or they can be extracted from data from simulated system traces. Some goal metrics that could be used are *goal model flexibility*, *goal criticality*, and *goal occurrence* in successful system traces. Goal model flexibility measures how well the model can adapt to failure of goal. Goal criticality ranks how essential a specific goal is to achieving the overall goal of the system. Goal occurrence measures how likely a goal will occur in a successful run of the system. These goal metrics would allow additional GMoDS properties to be defined. These properties will be defined in the specification model and then be implemented in the Execution Model.

There are several implementation details that need to be fine-tuned to allow designers to more easily use the GMoDS Execution Model implementation. The first is the separation of the specification and the code. The current implementation requires the specification to be coded in the application. The separation of the specification (as an XML file) and the code will allow a designer change the configuration without having to recompile the code. The second is creating a reasoning component that users can easily extend. The current Execution Model forces the designer to maintain a reference to the Instance tree. An Execution Model that allows coders to abstract away the details of maintaining this instance tree will make the designers job easier. The coder will only have to call the occurred method when an event occurs and then they will get a ChangeList back.

Also an implementation detail that has not been resolved is goal uniqueness. Goals in the system have two different ways of being unique. The definition of a unique goal (the one used in this thesis) is based on the uid of the goal, where if the uid of two goals is the same, those goals are the same goal. The uid of is assigned to an instance goal when the event happens. The second definition of uniqueness is based on the parameters and the specification goal. If the parameters and the specification goal are the same for two goals, then those goals are the same goal. These are both valid interpretations for defining a unique goal and we intend on providing a mechanism that will allow the designer to specify which type of uniqueness they would like to use in the system.

REFERENCES

1. Brown, G., Cheng B., Goldsby, H., Zhang, J. *Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems*, In Proceedings of the 2006 international Workshop on Self-Adaptation and Self-Managing Systems (Shanghai, China, May 21 - 22, 2006). SEAMS '06
2. Brown, G., Cheng, B., Goldsby, H., and Zhang, J. *Goal-oriented Specification of Adaptation Requirements Engineering in Adaptive Systems* International Conference on Software Engineering, 2006.
3. Burch, J., Clarke, E., and McMillan, K. *Symbolic model checking: 10^20 states and beyond*. Information and Computation, 98(2):142--170, June 1992.
4. Castro, J., Kolp M., and Mylopoulos, J. *Towards Requirements-Driven Information Systems Engineering*: Information Systems, 27(6):365--389, June 2002.
5. Cheong, C., Winikoff, M., *Hermes: Designing Goal-Oriented Agent Interactions*, AOSE 2005, LNCS 3950, pp. 16–27, 2006
6. Cheong, C., Winikoff, M., *Improving Flexibility and Robustness in Agent Interactions: Extending Prometheus with Hermes*, SELMAS 2005, LNCS 3914, pp. 189–206, 2006.
7. Clarke, M., Wing, J., *Formal Methods: State of the Art and Future Direction*, ACM Computing Surveys, 28:626--643, 1996.
8. Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. *Object-Oriented Development: The Method.* Prentice Hall International: Hemel Hempstead, England, 1994.
9. Compaq Computer, *Extended Static Checking for Java*, In: SIGPLAN Conf. on Prog. Lang. Design and Impl. (2002)
10. Darimont, R., and Lamsweerde, A. *Formal Refinement Patterns for Goal-Driven Requirements Elaboration*, ACM SIGSOFT Software Engineering Notes, Volume 21 , Issue 6, November 1996
11. Decomposition, http://en.wikipedia.org/wiki/Decomposition(computer_science), February 27, 2007
12. DeLoach, S., Zhong, C., *An Investigation of Reorganization Algorithms*, Proceedings of the International Conference on Artificial Intelligence (IC-AI'2006). June 2006, Las Vegas, Nevada, CSREA Press, 2006
13. Department Of Energy, *DOE Glossary*, http://www.directives.doe.gov/pdfs/doegeninfo/draft/glossary.pdf, February 27, 2007
14. Garlan, D., Shaw, M., *An Introduction to Software Architecture*. Technical Report. UMI Order Number: CS-94-166., Carnegie Mellon University. 1994
15. Goal, *http://en.wikipedia.org/wiki/Goal_(management)*, February 27, 2007
16. Holloway, M., *Why Engineers Should Consider Formal Methods*, Proceedings of the 16th AIAA/IEEE Digital Avionics Systems Conference, 1997
17. Holzmann, G. *The Model Checker SPIN*, IEEE Trans. on Software Engineering, 23(5):279--295, 1997.

18. Huget, MP. *Representing Goals in Multiagent Systems*, Proc. 4th Int'l Symp. Agent Theory to Agent Implementation(AT2AI–4), P. Petta and J. Mueller, eds., Austrian Soc. for Cybernetic Studies, 2004

19. Juan, T., Pearce, A., and Sterling, L. *ROADMAP: Extending the Gaia Methodology for Complex Open Systems*, Proceedings of the first international joint conference on Autonomous agents and multiagent systems, Bologna, Italy, ACM, 2002

20. Lamsweerde, A., Dardenne, A., Delcourt B., and Dubisy, F. *The KAOS Project: Knowledge Acquisition in Automated Specification of Software*. Proceedings AAAI Spring Symposium Series, Track: "Design of Composite Systems", Stanford University, March 1991, 59-62.

21. Lamsweerde, A., Darimont, R., and Letier, E. *Managing Conflicts in Goal-Driven Requirements Engineering* Software Engineering, IEEE Transactions on Volume 24, Issue 11, Nov 1998

22. Model Checking, *http://en.wikipedia.org/wiki/Model_checking*, February 27, 2007

23. Mylopoulos, J., Castro2, J., and Kolp1, M. *Tropos: A Framework for Requirements-Driven Software Development* Information System Engineering: State of the Art and Research themes, Brinkkemper, J., Solvberg, A. (eds), Lecture Notes in Computer Science, Springer Verlag, 2000.

24. Odell, J., Parunak, H. and Bauer, B., *Extending UML for agents*, Proceedings of Agent-Oriented Information Systems Workshop (AOIS-00), 2000.

25. Robby, Dwyer, M., and Hatcliff J. *Bogor: An Extensible and Highly-Modular Model Checking Framework*, Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering, September, 2003

26. Robinson W., *Integrating Multiple Specifications Using Domain Goals*, Proceedings of the 5th international Workshop on Software Specification and Design (Pittsburgh, Pennsylvania, United States). IWSSD '89.

27. Smith, D., *KIDS: a semiautomatic program development system*, IEEE Transactions on Software Engineering 16(9):10241043, September 1990.

28. Srinivas, Y., Jüllig, R., *SPECWARE: Formal Support for Composing Software*, In Mathematics of Program Construction, pages 399--422, 1995.

29. Taylo, D., and Procter M. *The Literature Review: A Few Tips On Conducting It.* http://www.utoronto.ca/writing/litrev.html, May 12, 2005.

30. Wooldridge, M., Jennings, N., and Kinny, D. *The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems*, Journal of Autonomous Agents and Multi-Agent Systems 3(3):285-312 (2000).

31. Writing Tutorial Services, Indiana University. *How To Write a Thesis Statement*. http://www.indiana.edu/~wts/pamphlets/thesis_statement.shtml. May 12, 2005.

32. Yu, E. and Mylopoulos, J. *Enterprise modelling for business redesign: the i* framework.* ACM SIGGROUP Bulletin archive, Volume 18, Issue 1 (April 1997)

33. Zambonelli, F., Jennings, N., Wooldridge, M., *Developing Multiagent Systems: The Gaia Methodology*, CM Transactions on Software Engineering and Methodology 12 (2003) 417470 137